移动开发系列

基于 Swift 的 Apple Watch 开发教程

王永超　编著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

Apple Watch 是苹果公司推出的智能手表，其系统为 watchOS，Swift 是苹果公司新推出的开发语言。本书介绍基于 Swift 3 进行 watchOS 3 应用的开发，对 Swift 3 基础、watchOS 3 基础、多媒体、游戏、传感器、GPS、与 iPhone 交互、健康监测等多项手表专属内容进行全面阐述和案例学习，可以更好地引导和帮助读者掌握新语言 Swift 和学习 Apple Watch 应用开发。

本书适合 iOS 开发、Swift 开发、Apple Watch 开发和学习，以及 Apple watch 爱好等众多读者。

图书在版编目（CIP）数据

基于 Swift 的 Apple Watch 开发教程 / 王永超编著. —北京：电子工业出版社，2017.9
（移动开发系列）
ISBN 978-7-121-32377-5

Ⅰ. ①基… Ⅱ. ①王… Ⅲ. ①程序语言－程序设计－教材 Ⅳ.①TP312

中国版本图书馆 CIP 数据核字（2017）第 183889 号

策划编辑：张 迪（zhangdi@phei.com.cn）
责任编辑：张 迪
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：710×1 000 1/16 印张：9.5 字数：243 千字
版 次：2017 年 9 月第 1 版
印 次：2017 年 9 月第 1 次印刷
定 价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）88254469；zhangdi@phei.com.cn。

前　言

Apple Watch 是苹果（Apple）公司推出的一款比较成熟的智能穿戴电子产品，具有运动追踪、健康监测、消息推送、多媒体、游戏、GPS 定位等多种功能，并支持第三方应用，如今已有较大市场占有率。Apple Watch 需要配合 iPhone 手机使用，通过经配对的 iPhone 访问应用商店进行第三方应用的下载和安装，Watch 上的应用都会包括一个手表端运行包和手机端运行包，分别运行在手表和手机上，并且两者之间可以交换数据。

Apple Watch 之所以被称为“智能手表”，是因为其上运行着智能操作系统 watchOS。watchOS 之所以被称为“智能操作系统”，笔者认为不是因为其可以连接手机并接受手机控制，而是因为其拥有应用程序的概念，为开发者提供了应用程序编程接口（API）、开发语言和开发编译工具 Xcode，并且允许开发者开发、部署和运行第三方应用程序（App）。watchOS 1 的应用运行在 iPhone 上并把运行结果发送到 watch 上进行显示。从 watchOS 2 开始，应用直接安装和运行在手表上，称为本地应用，又称原生应用（Native App），大幅提高了运行效率。

watchOS 允许开发者使用两种语言开发应用，分别是原有的 Objective-C 和新语言 Swift。Swift 是苹果公司推出的新开发语言，Swift 已经更新了多个版本，从 Swift 3 开始，Cocoa 类名去掉了 Objective-C 中的 NS 开头，直接定义为含义名称，例如 NSString 改为 String，NSTimer 改为 Timer，等等，并且类的成员也都有少量变化。Swift 也可以调用已有的 Objective-C 类进行混合编译。为了更好地推广 Swift，苹果公司已经将 Swift 开源化，允许开发者直接基于 C 语言开发 Swift 源码。Swift 具有语法简洁、易读、易写的特点，本书采用 Swift 开发 watchOS 应用，也为读者提供一个学习和使用 Swift 参考资料。

本书包括 8 章。第 1 章概述，介绍 watchOS 项目的结构组成，并列举了已经提供部分和完全支持的框架。第 2 章 Swift 编程基础，Swift 更新到第 3 个版本已经基本稳定，该部分介绍了 Swift 的基本语法、函数调用、常用数据结构和类，为后续开发做好准备。第 3 章 watchOS 基础，开始正式进入 watchOS 应用的开发，

介绍了页面和常用控件。第 4 章 watchOS 高级进阶，是第 3 章 watchOS 基础开发的进一步深入，主要涉及按压交互和组件交互，具体内容包括 Force Touch 菜单、振动引擎、表盘功能栏、提醒、后台刷新任务、URL 后台下载、Dock 截图、Apple Pay、通知等多种高级功能。第 5 章多媒体和游戏引擎，多媒体包括录音、连接蓝牙耳机播放音频、视频播放和喇叭外放，游戏引擎包括 2D 引擎 SpriteKit 和 3D 引擎 SceneKit，这里还结合游戏讲解了手势交互。第 6 章运动传感器和 GPS，Apple Watch 已经明确支持加速计、陀螺仪两种主要运动传感器，同时也支持 GPS 定位。第 7 章与 iPhone 交互，Apple Watch 不是 iPhone 的附属部件，而是运行独立操作系统的独立计算机，与 iPhone 之间进行平等的数据交换，watchOS 与配对的 iOS 交互是通过 WatchConnectivity 框架实现的，包括覆盖式后台传输、队列式后台传输、文件传输、消息传输、消息数据传输、功能栏传输等多个交互类型。第 8 章健康，Apple Watch 提供强大和全面的健康监测功能，如心率、步数、活动能量消耗等，同时会将监测到的数据发送到 iPhone 上的健康库中。上述的每一个部分在详细讲解的基础上，均提供了案例代码。

以 Apple Watch 为代表的智能穿戴产品还远远不如手机那么普及，存在巨大的市场增长空间，同时 Apple Watch 的应用和功能的开发还处在挖掘和探索阶段，而本书以此为契机，为广大开发者和其他兴趣读者学习和开发 Apple Watch 应用提供详细、系统的中文学习参考资料。笔者也衷心希望广大开发者早日能创造出几款杀手级应用。

目　录

第 1 章

概　　述

1.1 watchOS 项目结构

如今的 Apple Watch App 都要求是原生应用（native app），原生应用即是高于 watchOS 2 及以上的版本，并作为一个完整的应用包在 Apple Watch 上独立运行。本书介绍的手表操作系统版本是 watchOS 3，使用的语言版本是 Swift 3，如图 1-1 所示是 watchOS 3 App 的结构图。

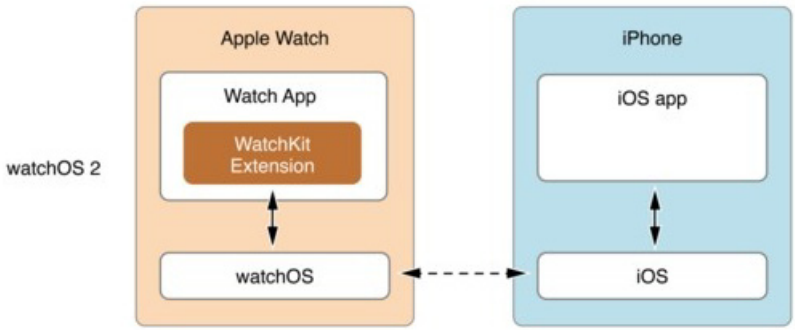


图 1-1　watchOS 3 App 结构图

从图 1-1 中可以看出，Watch App 作为一个独立应用在 watchOS 上运行，wachOS2/3 的 Xcode 项目包含 3 个部分，分别是 iOS App 包、Watch App 包和 WatchKit Extention 包。其中，iOS App 包负责 iPhone 端所有运行内容，Watch App 包包含界面编辑和手表应用整体参数，WatchKit Extention 包包括 watch 端运行的代码及资源。WatchKit Extension 包含在 Watch App 包中，而手表端 App 和手机端 App 通过操作系统（watchOS 和 iOS）进行交互。

1.2 建立 watchOS App 实例

（1）打开 xcode，新建项目，进入 watchOS 下的 Application，选择“iOS App with WatchKit App”，如图 1-2 所示。

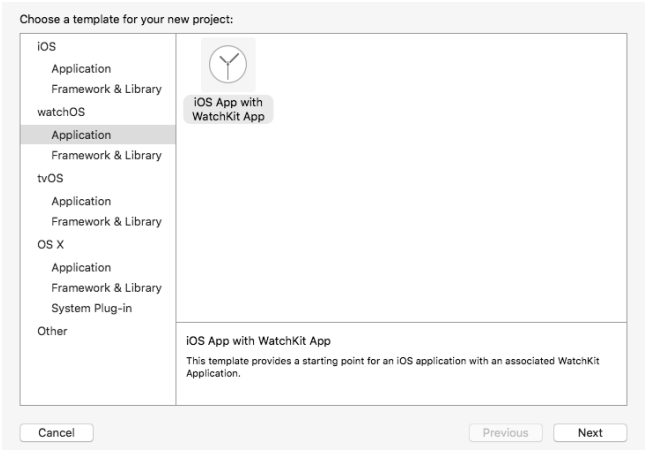


图 1-2　工程选择对话框

（2）单击 Next 按钮，设置项目名称为“NewApp”（其中 tinghe17 为笔者常用 id，与开发者账号有关的内容，如证书、签名、id 等内容不在本书范围之内），选择语言为 Swift，为了全面了解，下面选择通知场景（Notifaction Scene）、表盘功能栏（Complication），如图 1-3 所示。

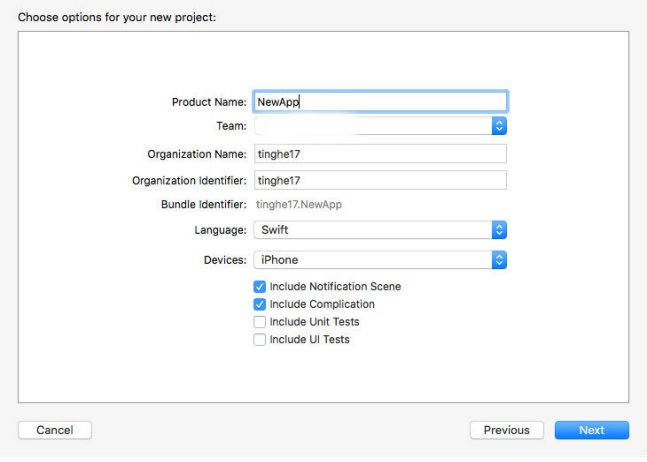


图 1-3　项目参数对话框

（3）单击 Next 按钮，选择保存路径（英文），项目建立成功。iOS App 包的名称为 NewApp，手表端包会自动起名字，即添加了后缀“WatchKit App”和“WatchKit Extension”。整个 NewApp 项目包含 3 个相应的 TARGETS。3 个 TARGETS 的 Bundle Identifier 存在前后的从属关系，依次为 tinghe17.NewApp、tinghe17.NewApp.watchkitapp 和 tinghe17.NewApp.watchkitapp.watchkitextension，如图 1-4 所示。

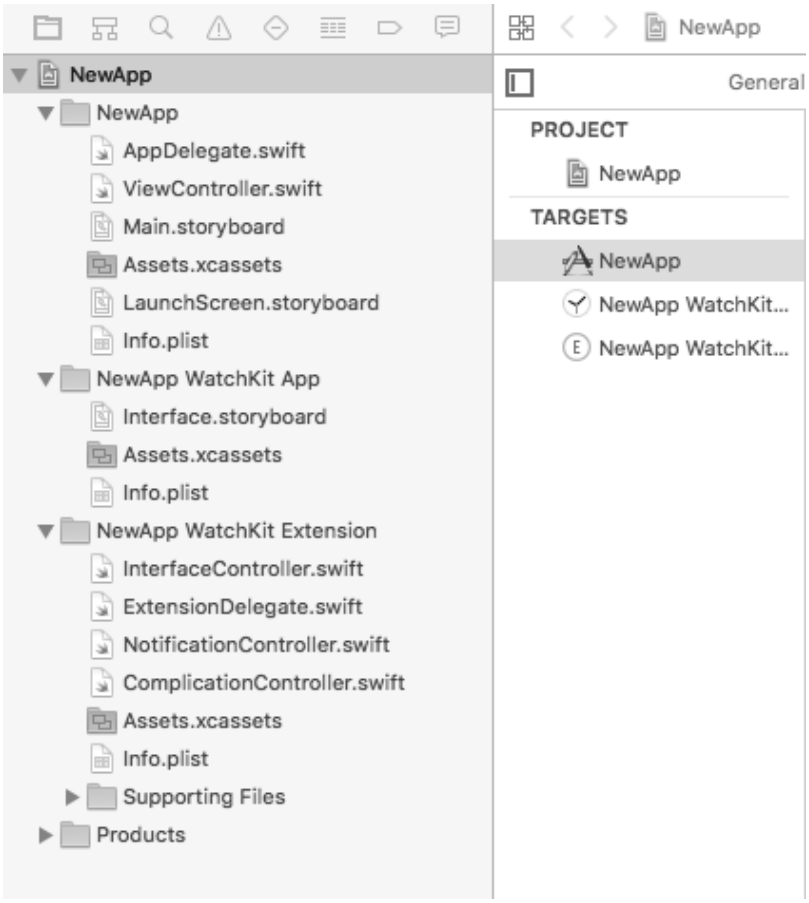


图 1-4　项目结构图

（4）在项目列表中选择 NewApp WatchKit App 中的 interface.storyboard，即可从主视图看到手表界面详细结构，包括一个应用运行时的界面（Interface Countroller）、一个静态通知界面（Static Interface）和一个由前者变化而显示的动态通知界面（Dynamic Interface），如图 1-5 所示。

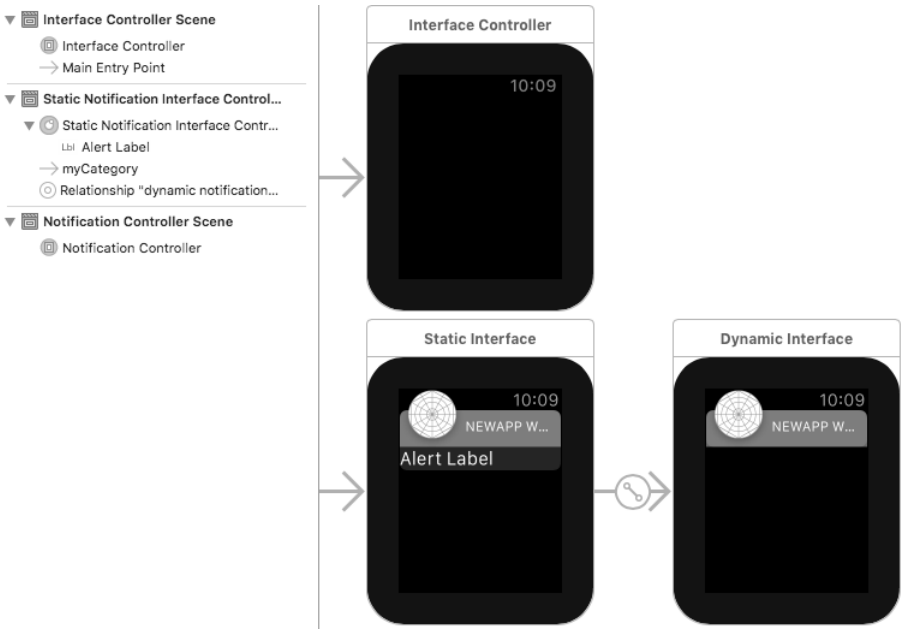


图 1-5　界面结构图

（5）选择调试项，包括默认的手机端 NewApp 和手表端的 WatchKit App、通知、与表盘功能栏。调试手表应用时选择手表端的 WatchKit App，并且选择适配的 Apple Watch 真机，如图 1-6 所示。

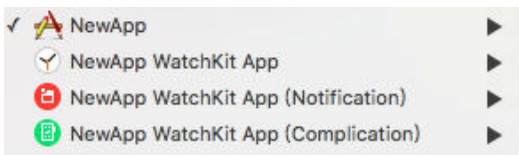


图 1-6　调试选项

1.3　watchOS 3 框架支持

watchOS 3 支持框架 framework 可以分为两类：一类是手表系统专属框架；另一类是 iOS 系统原有框架。其中，手表对于某些 iOS 系统原有框架是部分支持，同时某些原有框架加入了手表特有部分。本书重点介绍手表有关的框架及其内容。watchOS 3 支持的框架如表 1-1 所示。

表 1-1　watchOS 3 的框架支持

框架名	中文	框架名	中文
AVFoundation	音频视频	HealthKit	健康
ClockKit	表盘	HomeKit	家
Contacts	联系	ImageIO	图形输入输出
CoreAudio	音频	MapKit	地图
CoreData	数据	MobileCoreServices	移动服务
CoreFoundation	基础	PassKit	同行
CoreGraphics	图形	SceneKit	3D 游戏引擎
CoreLocation	定位	Security	安全
CoreMotion	运动	SpriteKit	2D 游戏引擎
CoreText	文本	UIKit	界面
EventKit	事件	UserNotifications	用户通知
Foundation	基础	WatchConnectivity	手表连接
GameKit	游戏	WatchKit	手表

1.4　【案例 1】watchOS 3 项目结构

watchOS 3 的手表端扩展包 Extension 包括 3 个文件：

- 页面，InterfaceController.swift；
- 扩展代理，ExtensionDelegate.swift；
- 通知界面，NotificationController.swift；
- 表盘功能栏，ComplicationController.swift。

（1）页面控件 InterfaceController 继承手表页面控件类 WKInterfaceController，默认生成三个函数，分别在不同的时机由系统调用：初始化时调用 awake（withContext context: Any?），页面显示时调用 willActivate()，不显示时调用 didDeactivate()。

```
override func awake(withContext context: Any?) {
       super.awake(withContext: context
    }
override func willActivate() {
       super.willActivate()
    }
override func didDeactivate() {
       super.didDeactivate()
    }
```

（2）扩展代理 ExtensionDelegate 继承手表扩展代理 WKExtensionDelegate，之所以叫“扩展代理”而不是“代理”，是因为该代理是 WatchKit Extention 的代理，而不是 WatchKit App 的代理。默认包含涉及手表应用声明周期：应用启动完成后初始化时调用 applicationDidFinishLaunching()，应用激活前台时调用 applicationDidBecomeActive()，应用失活时调用 applicationWillResignActive()，应用被系统后台启动时调用 handle（_ backgroundTasks: Set<WKRefreshBackgroundTask>）。

```
func applicationDidFinishLaunching() {}
func applicationDidBecomeActive() {}
func applicationWillResignActive() {}
func handle(
        _ backgroundTasks:Set<WKRefreshBackgroundTask>) {
           backgroundTasks.forEach { (task) in
                //处理代码
           task.setTaskCompleted()
       }
}
```

（3）通知页面控件 NotificationController 是在手表接到通知时单击通知按钮显示的页面。通知页面的声明周期与普通页面相似（同样包括初始化、激活、失活），只是多了一个接到通知的处理函数 didReceive()，默认该函数是标注不用状态，当使要显示通知页面动态显示内容时才须要调用该函数。

```
override func didReceive(_ notification: UNNotification,
                withCompletion completionHandler: @escaping
            (WKUserNotificationInterfaceType) -> Swift.Void) {
        completionHandler(.custom)
    }
```

（4）表盘功能栏控件 ComplicationController 负责表盘功能栏的设置，包括在 iPhone Watch 应用里表盘图库里的示例显示。ComplicationController 包含了多个函数，分别对应显示功能栏的不同时机，其中最常用的直接设置表盘功能栏显示普通状态的函数是 getCurrentTimelineEntry()，设置 iPhone Watch 里的功能栏示例的函数是 getLocalizableSampleTemplate()。

```
func getCurrentTimelineEntry(for complication: CLKComplication,
                    withHandler handler: @escaping
                (CLKComplicationTimelineEntry?) -> Void) {
        handler(nil)
        }
func getLocalizableSampleTemplate(for complication: CLKComplication,
                    withHandler handler: @escaping
                    (CLKComplicationTemplate?) -> Void) {
        handler(nil)
}
```

第 2 章

Swift 3 编程基础

Swift 作为一种新的编程语言，简洁易用，受到开发者的广泛喜爱。迄今为止 Apple 公司已经发布了 3 个 Swift 版本。本书在讲解手表开发之前，先介绍一下 Swift 3 语言的基本语法及其新特点。

2.1 基本语法

2.1.1 变量和常量

1. 普通变量

普通变量使用 var 声明，都有一定的初始值，例如：

```
//声明变量 a，初始值为 5，类型为整型 Int
var a=5
//声明变量 str，初始值为 abcd，类型为字符串 String
var str="abcd"
//声明对象变量 obj，类为 myClass
var obj=myClass()
```

2. 可空变量

可空变量是 Swift 特有的变量类型，其值可以为空（nil），而使用上节只用 var 声明的普通变量必须有一定的非空值。

```
//声明可空变量使用?
var a:Int?
//赋值跟普通变量相同
```

```
a=5
var aa=a //aa 也为 Int?类型
```

访问变量的值可以用!，!要求变量访问时不能是空，必须有值。

```
var b=a!
var c=a
//上边 b 值为 5:Int，c 值为 5:Int?
```

如果可空变量是类的实例 object，调用时可以使用!和?，!要求 object 不能是空，?表示可以为空，为空时不执行。

```
object=nil
object?.callFunction()//不报错，但不执行
object!.callFunction()//报错
```

3. 常量

常量用 let 声明，在之后的使用过程中其值不能变化。

```
let a=5
```

如果后面赋值 a＝8 就会报错

常量在后续使用中不使用!和?。

4. 数据转换

Swift 对数据格式要求很严格，所有数据转换都要求是明式的，在算数计算时要特别注意格式转换。

```
var a:Int=5
var b:Double=Double(a)
```

Swift 中还有一种特别的数据类型转换形式，即上下位类型转换，关键字为 as，如将上位的格式转换为下位的格式：

```
var son=father as Son
```

上述语句将父类变量 father:Father 转换为子类变量 son:Son。

如果 father 为可空变量 Father?，那么可添加“as!”转换为不可空变量，如：

```
var son=father as! Son
```

son 为 Son 类型。

也可使用“as?”转换为可空变量：

```
var son=father as? Son
```

son 为 Son?类型。

2.1.2 guard let 和 if let

“guard”的意思就是防护，顾名思义，作用就是确保条件判断为真，避免条件判断为假时的运行风险。guard 与 else 一起使用，如果 guard 后的语句判断为假，则执行 else 语句，else 语句必须为截断语句，如 fatalError()、return 等，防止继续执行代码出现错误。

guard 一般和 let、else 一起使用，声明一个临时变量并对该变量赋值，如果赋值不为空，则 let 声明语句为真，则该临时变量可供后续代码使用；如果赋值为空，let 声明语句为假，则进入 else 模块语句，截断继续代码执行，而 guard 的后续代码则不会被执行，let 声明的临时变量也不会使用了。一个 guard 语句可以同时多个 let 临时变量，使用逗号将 let 声明隔开即可。因为只有在可能是空的情况才需要 guard，所以 let 赋值语句等号右边的变量/常量必须为可空类型（?）的。

至于 if let {}没有 else，如果 let 语句为真，即 let 临时变量赋值非空，那么就执行后面的{}模块。

```
let a:Int?=nil //let a:Int?=5
guard a != nil else
{
fatalError("a is nil")
}
guard let b=a else
{
fatalError("a is nil")
}
```

```
print(b)//a is 5
if let c=a
{
print(c)// a is not nil
}
```

2.1.3　字符串 String

字符串 String 直接使用双引号，不用@。

1. 拼接

字符串拼接方式比较多，如符号“+”、插入方式“\()”、追加 append。

```
var string1="string0"+"string1"
var string2="string1 is \(string1)"
print(string2)//string1 is string0string1

//末尾添加，返回新字符串
var aString=string1.appending("Astring")
print(string1)//string0string1
print(aString)//string0string1Astring

//末尾添加到原字符串，不返回
aString.append("Bstring")
print(aString)//string0string1AstringBstring
```

2.长度

因为 String 兼容多种编码格式，所以获取长度时需要告知编码格式。另外，获取长度前也可以先获取字符顺序集合，再获取字符数量。

```
let length1=string1.lengthOfBytes(using: String.Encoding.utf8)
print(length1)//14
string1.characters.count
```

3. 包含判断

判断字符串是否包含另一个字符串，也可判断前缀或后缀是否是另一个字符串，返回布尔值。

```
//是否包含
string2.contains(string1)//true

//前缀
string2.hasPrefix("str")//true
//后缀
string2.hasSuffix("ing1")//true
```

4. 索引 Index

要访问字符串的字符和子字符串，需要用到字符索引 String.Index。字符索引代表字符在字符串中的位置编号，常用的是首位 startIndex、末尾 endIndex、偏移 offset 和索引范围 Range。范围 Range 的构建格式为 Range(index0..<index1)，表示区间[index0,index1]，包括两个端点。

5. 处理字符和子字符串

处理子字符串要通过索引 String.Index 来进行。

```
//插入字符
        string1.insert("1", at: string1.endIndex)
        print(string1)//string0string11

        //删除一个字符,返回删除的字符 Character
        let removedChar=string2.remove(at: string2.startIndex)
        print(removedChar)//s
        print(string2)//tring1 is string0string1

        //获取第一个"1"的位置
        let index1=string2.index(string2.startIndex, offsetBy: 5)

        //通过末尾索引获得子字符串
        let subString0=string2.substring(to: index1)
        //通过开始索引获得子字符串
        let subString1=string2.substring(from: index1)
        print(subString0)//tring
        print(subString1)//1 is string0string1

        //构造范围,注意不包括 index2
```

```
        let index2=string2.index(string2.endIndex, offsetBy: -7)//0
的索引
        let range=Range(index1..<index2)

        //截取字符串，通过范围获得子字符串
        let subString2=string2.substring(with: range)
        print(subString2)//1 is string0

        //替换原字符串,不返回
        string2.replaceSubrange(range, with: "replaceSubrange")
        //替换返回新字符串，原字符串不变
        let string3=string2.replacingCharacters(in: range, with:
"replacingCharacters")
        print(string2)//replaceSubrange1 is string0string1
        print(string3)//replacingCharactersceSubrange1 is string0string1
```

6. 分组

字符串可以依照特定字符串分割成字符串数组。

```
//分割成数组
let strings=string3.components(separatedBy: "  ")//根据空格分后三组
print(strings[0])//replacingCharactersceSubrange1
print(strings[1])//is
print(strings[2])//string0string1
```

2.1.4　数组

数组在应用开发时是必不可少的，本书介绍一下 Swift 数组的基本语法。

1. 数组声明

声明一个整形的一维数组。

```
var dataArray0=[Int]()//空数组
var dataArray1=[1,2,3,4,5]//含有 5 个初始值
//初始值含有 3 个 5
var dataArray2=[Int](count:3,repeatedValue:5)
```

声明二维空数组数组，多维数组类似。

```
var dataArray3=[[Int]]()
```

2. 属性和方法

数组具有多个属性和方法，方便我们使用数组，详见表 2-1（表中 dataArray 为数组对象）。

表 2-1　数组的属性和方法

属性/方法	说明
dataArray.count	包含元素数量
dataArray.first	第一个元素，索引为 1
dataArray.last	最后一个元素
dataArray.append(5)	添加一个元素 5，“+=“也可以
dataArray.insert(8,atIndex:2)	在索引 2 的位置插入元素 8
dataArray.removeAll()	清空
dataArray.romoveAtIndex(1)	删除索引 1 的元素
dataArray.romoveFirst()	删除首元素
dataArray.romoveLast()	删除尾元素

2.1.5　枚举

枚举 enum 就是预先设定几个字符串（值）构建一个数据类型，这个数据类型的实例的取值只能是预先设定的字符串之一，不能是其他值。如果已经知道变量的枚举类型，可以直接使用“.值”的方式表示枚举值。

```
//构建枚举类型 AppleProducts
enum AppleProducts{
    case iPhone
    case iPad
    case AppleWatch//单项要连续，不能有空格
    case MackBook,AppleTV//可以一行写两个
}
var ap=AppleProducts.AppleWatch
       ap = .iPhone//ap 的类型已知，则可以直接用枚举值.//不用写类型
       if ap == .AppleTV
       {
           //处理代码
       }
```

2.1.6　for 循环

for 循环是最常用的遍历语句，在新版的 Swift 中使用 in，具体语法如下。

```
var dataArray=[Int](count:3,repeatedValue:5)
//遍历下标
for i in 0..<dataArray.count
{
print(dataArray[i])
}
//不需要使用下标直接遍历
for data in dataArray
{
print(data)
}
```

上述循环中，i 的初始值为 0，通过“..”自己增加，最终到“< dataArray.count”（count-1）截止。之前 for 循环常用的运算符++在 Swift 新版本中已经废除。

2.1.7　switch/case 多条件判断

switch-case 语句是常用的多条件判断语句，每个 case 下面要求必须包括至少一句执行命令，不用写“break”可以自动跳出，如果 switch 判断的是枚举，且枚举的所有值都被 case 罗列出来，那么就不用写最后的 default 模块了。

```
//前续枚举部分
switch ap
{
case  .iPhone, .AppleWatch://多种情况
   print(ap)
case .AppleTV:
   print(ap)
case .iPad:
   break
//case .MacBook:break//若写.MacBook 就不用 default 了
default : print(ap)
}
```

2.1.8 任意类 Any

Any，即任意类，可以将任意一个其他类的对象与 Any 对象相互转换，在转换时使用 as。常用的数值类型 Int 和 String 等可以在要求 Any 类的地方省略 as 转换，直接赋值。在传递参数时使用 Any 表示不限制于参数的类。动态字典元素值的类型默认为 Any。

2.2 函数

2.2.1 声明和调用

函数在面向对象的开发方式中一般写成类的方法，声明时使用关键字 func。

声明一个单参数方法，没有返回值（->Void 可以省略）。

```
func opration(a:Int)->Void
{
print(a)
}
```

声明一个多参数方法，有返回值。

```
func sumabc(a:Int,b:Int,c:Int) -> Int
{
let s=a+b+
return s //必需要返回一个 Int，否则报错
}
```

调用函数使用方法名和形参，调用单参数函数。

```
opration(a:5)
```

调用多参数函数时要写上所有定义函数时用的形参。

```
var sum=sumabc(a:1,b:2,c:3)
```

2.2.2　回调函数

回调函数在 Apple Watch 编程中会经常使用，尤其是在调用传感器接口的时候。回调函数是一个函数执行块，将其传入另一个函数的参数中，回调函数不会立刻执行，而是会在某个特定时机被调用执行（即异步的），如得到传感器返回的数据时。回调函数的参数在声明时表示要传出的数据，而在调用执行时代表捕捉到的数据。在执行时如果访问类成员时要用 self。

在 Swift 中要注意回调函数的语法格式。

带有回调函数的声明形式如下。

```
OperationFunction( a:Int,
                handler:(Int, Double)
                ) -> Void
{
handler(a,2*a)//将 a 和 2a 传出去
}
```

上面 handler 是回调处理模块。

在调用 OperationFunction 时按照如下两种语法格式书写：

（1）格式一

```
OperationFunction(a:9)
{(b, c)  in
print(b)
print(c)
}
```

输出为如下。

```
9
18
```

（2）格式二

```
OperationFunction(a:9,
handler :{(b, c) in
print(b)
print(c)
})
```

定义回调模块的具体执行内容使用关键字“in”。在调用函数 Operationfunction 时，会先根据传入参数 9 进行计算处理（处理过程不需要知道），然后在特定条件时（异步）返回两个 b:Int 和 c:Double 参数（变量名字为自定义），并将其传入回调函数 handler 中进行计算，handler 里的具体执行代码是我们自定义的处理代码。

2.2.3 异常抛出和捕捉

某些代码（如读取文件）执行时会抛出（throws）异常，必须进行捕捉。捕捉异常使用 do-try-catch 代码块。如果不进行捕捉，则可用“try!”，代码比较简洁，但如果出错会使应用崩溃。

```
do{
    try obj.function() // obj.function()可能会抛出异常
  }catch{
          //处理异常
 }
try! obj.function()
```

2.3 常用数据结构和类

2.3.1 字典 Dctionary

字典 Dctionary 是常用的数据结构之一，字典是否可变取决于声明管检测，使用 var 声明的字典可变，使用 let 声明的字典不可变。

1. 字典的创建

字典的每个元素都包含一个键（key）和值（value），创建时 key 和 value 都可以确定数据类型，也可以不设定，不设定时数据类型为 Any，所以有时保存类

的对象需要用 as 转为 Any。

```
// 创建字典变量，指定数据类型为 Int 型
let dic1 = [String:Int]()

// 创建字典变量，Dictionary<String, Int> 等价于 [String:Int]
let dic2 = Dictionary<String, Int>()

// 创建字典变量，不指定数据类型
let dic3:Dictionary = [ " key1 " :1, " key2 " :2]

// 创建字典变量，指定数据类型为 Int 型
let dic4:Dictionary<String, Int> = [ " key1 " :1, " key2 " :2,
" key3 " :3]

// 创建字典变量，不指定数据类型
let dic5 = [ " key1 " :1, " key2 " :2, " key3 " :3, " key4 " :4]

// 创建指定数据类型的字典
let dic6:[String:Int] = [ " key1 " :1, " key2 " :2, " key3 " :3,
" key4 " :4]

// 创建指定数据类型的字典
let dic7:[String:Any] = [ " key1 " :1, " key2 " :2, " key3 " :
" value3 ", " key4 " :4]
```

2. 修改增加键值

修改键值可以直接赋值，也可以通过 updateValue ()函数修改；若原无此键，则视为增加键和键值。

```
var dic:Dictionary = [ " key1 " :1, " key2 " :2, " key3 " :3, " key4 " :4]
// key 不存在时，追加键值对（key 值为新增的，若 key 值已存在则为修改对应的
value 值）
dic[ " key5 " ] = 5
dic.updateValue(6, forKey: " key6 " )
```

3. 删除

```
var dic = [" key1 " :1, " key2 " :2, " key3 " :3, " key4 " :4]
// 删除指定 key 对应的值
dic.removeValue(forKey: " key2 " )
// 删除字典的所有元素
dic.removeAll()
```

4. 元素个数

元素个数可以直接访问 count 属性。

```
let num:Int = dic.count
```

5. 访问 key 和 value

通过 key 可以访问 value，也可以得到所有 key 的集合 keys 和所有 value 的集合 values，两个集合可以排序和遍历。

```
var dic:Dictionary = [" key1 " :1, " key2 " :2, " key3 " :3, " key4 " :4]
        // 获取字典中指定 key 对应的值
        let value1 = dic[" key1 " ]
        //遍历值
        for k in dic.keys.sorted() {
            print(k)
        }
       //遍历值
        for v in dic.values.sorted() {
            print(v)
        }
        //输出
        //key1
        //key2
        //key3
        //key4
        //1
        //2
        //3
        //4
```

```
//根据键遍历值，这里没有排序
for key in dic.keys {
            print(dic[key])
        }
```

2.3.2　日期 Date

Date()创建的是当前日期，我们还可以通过日历 Calendar 和日期构成 DateComponents 来构造出指定的某个日期。时间 Date 可以通过 DateFormatter 格式化成字符串。Date 的两个实例先日期、后日期和间隔，三者可以由二得一。

```
    //获取当前日历
    let calendar = Calendar.current
    //日期组成
    var dateComp  = DateComponents()
    //设置日期各个部分的值
    dateComp.year = 2016
    dateComp.month = 7
    dateComp.day = 3
    dateComp.hour = 0
    dateComp.minute = 0
    dateComp.second=0
    //创建成指定的日期
    let date=calendar.date(from: dateComp)
    //创建一个日期格式器
    let dateFormatter = DateFormatter()
    //设置显示格式
    //formatter.dateStyle = .none//无
    //formatter.timeStyle = .long//长
    dateFormatter.dateFormat = "yyyy:MM::dd HH:mm:ss"
    //全格式，可以根据需要选择要显示的部分，如"dd-HH:mm"
     let dateString = dateFormatter.string(from: date!)
    //当前日期
    let date0=Date()
    //30 秒后的日期
```

```
let date1=date0.addingTimeInterval(30)
//获取当前日历
let calendar = Calendar.current
//日期组成
var dateComp  = DateComponents()
//设置日期各个部分的值
dateComp.year = 2016
dateComp.month = 7
dateComp.day = 3
dateComp.hour = 0
dateComp.minute = 0
dateComp.second=0
//创建成指定的日期
let date=calendar.date(from: dateComp)
//创建一个日期格式器
let dateFormatter = DateFormatter()
//设置显示格式
//formatter.dateStyle = .none//无
//formatter.timeStyle = .long//长
dateFormatter.dateFormat = "yyyy:MM::dd HH:mm:ss"
//全格式，可以根据需要选择要显示的部分，如"dd-HH:mm"
 let dateString = dateFormatter.string(from: date!)
//当前日期
let date0=Date()
//30 秒后的日期
let date1=date0.addingTimeInterval(30)
//两个日期的间隔
let interval=date1.timeIntervalSince(date0)
//值为 30
//比较两个日期
let result=date1.compare(date0)
switch result
{
case .orderedAscending:
```

```
    print("升序")
    case .orderedDescending:
    print("降序")
    case .orderedSame:
    print("相同")
    }
```

2.3.3　计时器 Timer

计时器 Timer 可以每隔一段时间执行一个函数，多用于计时、等待、数据刷新和界面刷新等任务。

```
let myObject=MyObject()
mdic.setObject(myObject, forKey:"myObjectName" as NSCopying)
mdic["myObjectName"]= myObject //这里不需要 as
//计时器/类成员
var timer:Timer?
//开启计时器，3 秒钟重复循环
timer=Timer.scheduledTimer(timeInterval: 3, target: self, selector:
#selector(InterfaceController.updateTimer), userInfo: nil, repeats:
true)
```

```
//#selector 也可以写为如下格式
     //timer=Timer.scheduledTimer(timeInterval: 3, target: self, selector:
#selector(updateTimer(uTimer:)), userInfo: nil, repeats:true)
//timer?.fire()//不 fire 则 3 秒之后触发
//循环执行函数
func updateTimer(uTimer:Timer)
{
//执行代码
}
//停止
timer?.invalidate()
```

2.3.4 文件存储

由于 watch 应用结构是 app+extension 结构， document 文件夹容易造成无法读写的问题，所以 watch 应用的数据文件要存储在组目录中。对文件的读写等非内存操作都会抛出异常，需要使用 do-try-catch 语句进行异常捕捉。文件路径 path 和 URL 基本上可以通用，都有对应的函数。

```
//group 地址
let groupURL= FileManager.default.containerURL(
forSecurityApplicationGroupIdentifier: "group.tinghe17.AudioVideo")

//文件地址
let fileURL=groupURL?.appendingPathComponent("recordFile.mp4")

//目录和文件操作
let documentPath =NSURL.init(fileURLWithPath:  "myfolder",
             isDirectory: true, relativeToURL: groupURL)
```

```
do {
            //创建目录
          try fileManager.createDirectoryAtPath(documentPath!,
withIntermediateDirectories: true, attributes: nil)
          let filename= "myFileName.ext"
              let savefilepath=documentPath!+"/"+filename
              if fileManager.fileExistsAtPath(savefilepath)
              {
                 //删除原来的
                  try fileManager.removeItemAtPath(savefilepath)
              }
              //移动文件
              try fileManager.moveItemAtPath(file.fileURL.path!,
toPath: savefilepath)
       }catch {
       }
```

2.4 Objective-C 混合编程

在 Swift 出来之前，苹果开发语言是 Objective-C，所以到现在为止积累了大量的 Objective-C 资源，尤其是算法，希望可以继续使用。所以苹果研发的 Swift 是可以与 Objective-C 兼容的，这里就介绍一下使用 Swift 开发 watchOS 应用时加载 Objective-C 混合编程的方法，混合编程的原理是通过一个“桥”头文件(“bridging header”) 集中地将 Objective-C 代码暴露该编译器，就可以在 Swift 环境中以 Swift 的语法形式访问和使用这些 Objective-C 代码里的类或接口了。本书示例中提供了一个混合编程的例子“MixedProgramming”，混合编程具体步骤如下。

（1）将 Objective-C 的 h 文件和 m 文件添加到 Xcode 项目的 WatchKit Extension 目录下。注意，不是 WatchKit App 下。

（2）如果添加的是 Objective-C 的单独文件，Xcode 会提示添加“桥“头文件（名称后缀一般为“Bridging-Header.h”）。如果是文件夹，则不会提示，那么就需要我们自行创建“桥”头文件。

（3）检查项目 TARGETS 中 WatchKit Extention 的“Build Settings”设置，确认“Enable Modules（C and Objective-C）”值为“Yes”。

（4）确保 Build Settings 的“Objective-C Bridging Header”值为桥头文件相对于 WatchKit Extention 项目的相对路径。

（5）在头文件中引入（#import）需要访问的 Objective-C 头文件。

第 3 章

watchOS 基础开发

3.1 页面控件

3.1.1 页面生命周期

页面控件 WKInterfaceController 是手表应用屏幕最底层的整个视图窗口控件，相当于手机中的 UIViewController，App 的 Interface.storyboard 中每个页面都要对应 Extension 中的一个页面类，页面类 WKInterfaceController 默认有 3 个方法，分别对应其生命周期的 3 个时间点。

1. 创建和初始化

WKInterfaceController 最开始进行创建，然后进行配置要显示的内容，这个步骤对应方法 func awake(withContext context: Any?)，在此方法中进行页面初始化，设置页面显示需要的参数。此函数在页面整个生命周期中只在开始时执行一次。context 是创建新的页面时传入的 Any 类型参数，供页面显示调用。

2. 激活

当一个页面要显示在当前屏幕时，会调用 func willActivate()，此函数在 awake() 函数后执行，并且在整个生命周期中每当页面被切换到前面时都会被执行。

3. 失活

页面被切换、被遮挡、被关闭，则会激发失活方法 func didDeactivate()。

3.1.2 页面关系

不同的页面之间具有两种关系，一种是左右并列显示关系，另一种是前后遮挡关系。

1. 左右并列

左右并列是 Apple Watch 最常用的页面组织方式，用手指在手表上左右滑动就可以在不同页面之间切换。屏幕底部中间会有与页面数量一致的标志点。

在 xcode 中按住 control 键，用鼠标左键从一个页面控件拉线到另一个页面控件，在弹出的菜单上选择“Relationship Segue:next page”，连接好后如图 3-1 所示。

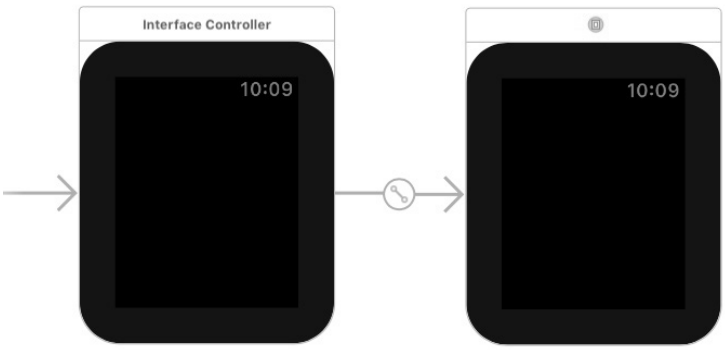


图 3-1　页面并联关联

2. 前后遮挡

前后遮挡就是弹出一个新的页面，把原来的页面遮挡住，原来的页面虽然看不到但还是保留在内存中。新弹出页面左上角是关闭按钮，新页面关闭后会显示出原来的页面。调用新页面使用函数：

```
self.presentController(withName: String, context: Any?)
```

其中，withName 是页面在 Interface.storyboard 中的标志 identifier，context 是传入的参数，供新页面在函数 func awake(withContext context: Any?)函数中使用。Any 表示传入的参数可以是数值、字符串，也可以是类的对象，注意访问参数时要用 as!转换成正确的数据类型。

3.2 常用控件

3.2.1 表盘布局和 Group

控件在表盘的位置是有限制的，不能设置为任意位置。表盘布局为上中下三

行，每一行又分为左中右三个位置，使用组控件 Group 可以嵌套排版，如果内容过多，编辑器中的表盘页面还可以向下拉伸。组控件 Group 既可以横向排版，也可以竖向排版。

3.2.2 图片

图片资源应放在 WatchKit App 中的资源文件夹 Assets.xcassets 中，不能放在 Extension 的资源文件夹中。

图片控件 WKInterfaceImage 可以用来显示图片，在 storyboard 中可以设置其位置、大小和缩放形式。

```
//拉线建立成员变量
@IBOutlet var imageView: WKInterfaceImage!

//指定图片
imageView.setImageNamed("ImageName")
```

3.2.3 按钮

在 storyboard 的页面中拖动创建按钮，设置标题、位置和大小，然后拉线到代码中，一般只需要链接事件方法即可，如果要改变标题，也可以拉线创建按钮变量。

```
//拉线创建的按钮单击事件方法
@IBAction func onDetailButton() {
        //处理代码
    }
```

3.2.4 开关

开关 Switch 具有打开和关闭两种状态，拉线建立成员变量，并拉线建立开关事件。

```
@IBOutlet var showSwitch: WKInterfaceSwitch!
@IBAction func changeSwitch(_ value: Bool) {//处理代码 }
```

3.2.5　滑动条

滑动条具有当前值、最小值、最大值和步数。同样地，拉线建立成员变量，并拉线建立开关事件。

```
@IBAction func onSlider(_ value: Float) {
    //处理代码
}
```

3.2.6　选择器

选择器的类名为 WKInterfacePicker，在 storyboard 页面的适当位置添加 Piker 控件，然后拉线到代码连接变量和选择事件方法，同时设置显示参数，如焦点框。

1. 配置选择项

使用选择器时，需要预先设置显示的选择项，选择项为 WKPickerItem，每一项可以包括标题 title:String 和图标 contentImage:UIImage。

```
//（全局）预设字典
let namesArray=["边境牧羊犬","贵宾/泰迪","德国牧羊犬","金毛巡回犬","杜宾"]
//配置 5 个选择项
let pikerItem0=WKPickerItem()
 pikerItem0.title=namesArray[0]
 let pikerItem1=WKPickerItem()
 pikerItem1.title=namesArray[1]
 let pikerItem2=WKPickerItem()
 pikerItem2.title=namesArray[2]
 let pikerItem3=WKPickerItem()
 pikerItem3.title=namesArray[3]
 let pikerItem4=WKPickerItem()
 pikerItem4.title=namesArray[4]
```

```
//将选择项添加到选择器中
//@IBOutlet var myPicker: WKInterfacePicker!
myPicker.setItems([pikerItem0,
```

```
pikerItem1,
pikerItem2,
pikerItem3,
pikerItem4])
```

2. 选择事件

旋转表冠，可以操作具有当前焦点的选择器进行滚动，从而激活选择事件方法，系统给提供当前选择项的序列号 value。

```
@IBAction func onMyPicker(_ value: Int) {
        //处理代码
}
```

3.2.7 表格

表格 WKInterfaceTable 可以显示多行类型相同的内容，每一行的内容都需要在代码中设置，并且还要自定义行的标识 identifier 和类。跟手机列表不同的是，手机列表不接受触控选择事件，所以如果逻辑功能需要处理选择事件时，我们在行中添加按钮来响应事件即可。

1. 添加表控件

在 storyboard 界面，从控件栏拖动一个“Table”控件到一个页面中，显示为一行。再拖曳一个按钮控件到行中，如图 3-2 所示。

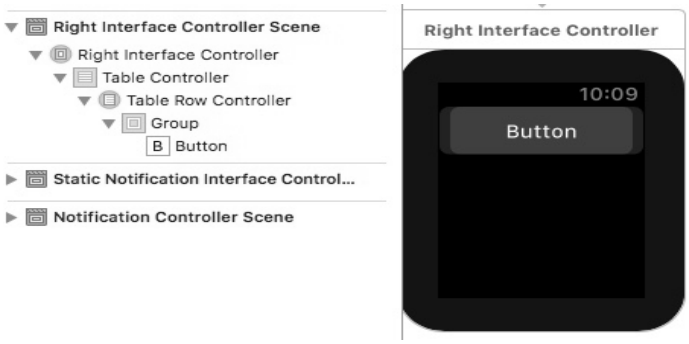


图 3-2　Table 行添加按钮控件

2. 新建行类

要在代码运行时设置行的内容，必须要自定义一个代表行的类，在 storyboard

里设置行关联建立的类，设置行的标志 identifier 为“myRowID”，并拉线建立按钮变量和事件方法，如图 3-3 所示。

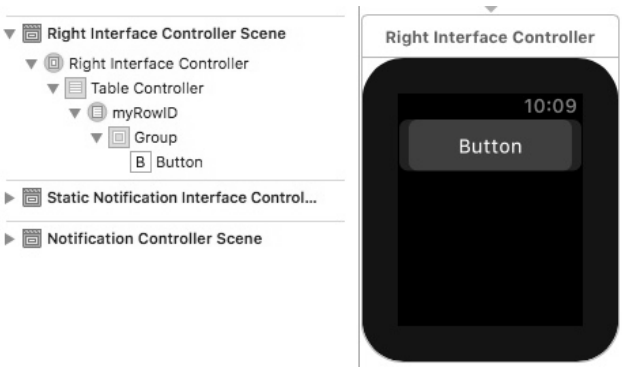


图 3-3　Table 建立行类

```
class MyRowController: NSObject {
      var index=0
    @IBOutlet var rowButton: WKInterfaceButton!
    @IBAction func onRowButton() {
    }
}
```

3. 设置表格内容

在页面的代码中设置表格的行数，以及每行的序列号和内容。

```
//设置表格内容
func setTable()
{
   //设置表格行数和行的类型 id
   let rowCount=5
   tableController.setNumberOfRows(rowCount, withRowType: "myRowID")
    //设置每行内容
    for i in 0..<rowCount
    {
       let row=tableController.rowController(at: i) as! MyRowController
       row.rowButton.setTitle(namesArray[i])
       row.index=i
```

```
        }
}
```

4. 配置触摸事件

我们现在设计触摸事件为弹出新界面，因为弹出界面需要 WKInterfaceController 的对象来执行 presentController 方法，所以需要建立一个全局的现实表格页面对象执行显示新页面的方法。

```
//声明全局变量
var  rightInterfaceController:WKInterfaceController?
//在页面中把当前页面传给全局变量
rightInterfaceController=self
//在行的按钮方法中显示新页面
@IBAction func onRowButton() {
rightInterfaceController?.presentController(withName:    "detail",
context: index)
}
```

3.3 应用图标

为了方便读者应用图标，这里把手机和手表应用涉及的图标及格式列出来。图标为不含透明 alpha 通道的 png 图片，根据需要的图片显示倍数确定名字@后的数字，如果是 2 倍，则命名为“filename@2x.png”，如表 3-1 所示。表栏功能栏 Complication 的初始图标则没有像素尺寸限制，程序运行后显示图片的显示尺寸和像素大小在后续章节会详细介绍。

表 3-1　应用图标格式

显示尺寸	位置	中文	倍数	像素尺寸
20pt	iPhone Notifacation	手机通知	@2x	40×40
			@3x	60×60
29pt	iPhone Settings	手机设置	@2x	58×58
			@3x	87×87
40pt	iPhone Spotlight	手机搜索	@2x	80×80
			@3x	120×120

（续表）

显示尺寸	位置	中文	倍数	像素尺寸
60pt	IPhone App	手机桌面图标	@2x	120×120
			@3x	180×180
24pt	38 Watch Notification Center	38mm 手表通知中心	@2x	48×48
27.5pt	44 Watch Notification Center	44mm 手表通知中心	@2x	55×55
29pt	Watch Companion Settings	手表配对设置	@2x	58×58
			@3x	87×87
40pt	Watch Home Screen(All)	表盘图标（38 和 42）	@2x	80×80
86pt	38mm Watch Short Look	38mm 手表通知短视图	@2x	172×172
98pt	42mm Watch Short Look	42mm 手表通知短视图	@2x	196×196

为方便做图，把所有图标按照像素排重后由大到小顺序排列后如表 3-2 所示。

表 3-2　图标图片的像素尺寸

像素尺寸	倍数	显示尺寸	像素尺寸	倍数	显示尺寸
196×196	@2x	98pt	80×80	@2x	40pt
180×180	@3x	60pt	60×60	@3x	20pt
172×172	@2x	86pt	58×58	@2x	29pt
120×120	@2x	60pt	55×55	@2x	27.5pt
	@3x	40pt	48×48	@2x	24pt
87×87	@3x	29pt	40×40	@2x	20pt

3.4　【案例 2】宠物乐园

本案例主要使用上一节介绍的几个控件构建一个应用，这个应用先显示一个表格，列出 5 种常见的宠物狗，单击某个名字就会弹出新界面对选择的品种使用图片和文字进行介绍。

1. 项目文件结构

手表端项目主要包括3个文件，“InterfaceController.swif”作为主页面使用Piker显示宠物；“RightInterfaceController.swift”为与主页面并列的右侧页面，采用表格的方式显示宠物，“MyRowController.swift”是表格行类，对应每行的内容；“DetailInterfaceController.swift”显示单个宠物的详细信息。项目文件结构如图 3-4 所示。

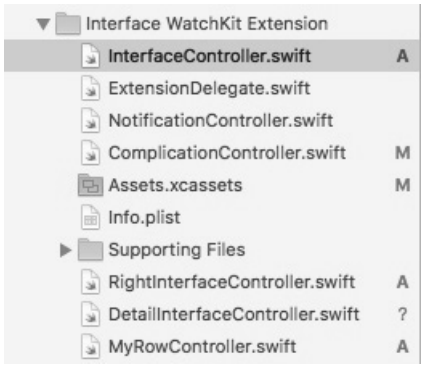


图 3-4　项目文件结构

2. 页面组成

与文件结构对应的，界面也包括 3 个，Xcode storyboard 中的显示如图 3-5 所示。

图 3-5　项目页面

3. 开发流程

主页面负责使用 Picker 的形式让用户选择显示的宠物，单击下面的“详情”按钮弹出选择宠物详情的页面。

1）设置宠物资料

本应用涉及 5 种宠物，包括宠物的名字、图片和文字描述。先将宠物图片加载到 WachKit App 的资源包 Assets.xcassets 中，然后在 InterfaceController.swift 中添加全局的名字、图片和文字描述。

```
    //全局预设字典
    let namesArray=["边境牧羊犬","贵宾/泰迪","德国牧羊犬","金毛巡回犬","杜宾"]
    let imageNameArray=["bjmyq","td","dgmyq","jmxhq","db"]
    let descriptionArray=["边境牧羊犬（Border Collie）原产于苏格兰边境，为柯利牧羊犬之一种，具强烈的牧羊本能，天性聪颖、善于察言观色，能确实明白主人的指示，可借由眼神的注视而驱动羊群移动或旋转，被当成牧羊犬已有多年的历史。其最大的特点是聪明、学习能力强、理解力高、容易训练、温和、忠诚、顺从，其忠心程度可以用如影随形来形容。",
    "贵宾犬也称"贵妇犬"，属于非常聪明且喜欢狩猎的犬种，据猜测贵妇犬起源于德国，在那儿它以水中捕猎犬而著称。然而许多年以来，它一直被认为是法国的国犬，贵宾犬分为标准犬、迷你犬、玩具犬三种。它们之间的区别只是在于体型的大小不同。我们所说的"泰迪"，其实就是贵宾。",
    "德国牧羊犬（German Shepherd Dog），是狗的一个品种，原产德国，具体的血统来源不明。唯一能够确认的就是，于 1880 年此犬已经在德国各地固定下来，并作为牧羊犬使用。他们敏捷且适合动作式的工作环境，它们经常被部署各种任务。后于第一次世界大战期间被德军募集，作为军犬随军。由德军取长补短培育后，基本定型。因为体型高大，外观威猛，并且具备极强的工作能力，因此在全世界范围以警犬、搜救犬、导盲犬、牧羊犬、观赏犬及家养宠物犬等身份活跃。",
    "金毛犬，属于均称、有力、活泼的一个犬种，特征是稳固、身体各部位配合合理，腿既不太长也不笨拙，表情友善，个性热情、机警、自信。因为金毛犬是一种猎犬，在困苦的工作环境中才能表现出它的本质特点。金毛犬总体外貌、平衡、步态和该犬种的用途应得到比身体任何一个单独部分更多的重视，除此之外，金毛与人类的情趣生活离不开关系了。",
    "杜宾犬，即笃宾犬。原产德国，它是根据培育这一品系人的名字路易斯·杜伯尔曼先生命名的，是所有品系中最富智慧的一种。杜宾犬是军、警两用的犬只。"]
```

2）置主页面

参照图 3-5，在主页面上添加控件，然后在 InterfaceController.swift 文件中添加 Picker 的代码。Picker 在多种宠物中切换时，下面的显示对应宠物的名字，并且单击“详情按钮”会弹出对应宠物的详情页面。

```
    @IBOutlet var myPicker: WKInterfacePicker!
    var pikerSelectedIndex=0
    @IBOutlet var nameLabel: WKInterfaceLabel!
```

```
override func awake(withContext context: Any?) {
       super.awake(withContext: context)
       // Configure interface objects here.
       //设置选择项
       let pikerItem0=WKPickerItem()
       pikerItem0.title=namesArray[0]
       let pikerItem1=WKPickerItem()
       pikerItem1.title=namesArray[1]
       let pikerItem2=WKPickerItem()
       pikerItem2.title=namesArray[2]
       let pikerItem3=WKPickerItem()
       pikerItem3.title=namesArray[3]
       let pikerItem4=WKPickerItem()
       pikerItem4.title=namesArray[4]

     myPicker.setItems([pikerItem0,pikerItem1,
     pikerItem2,pikerItem3,pikerItem4])
       nameLabel.setText(namesArray[pikerSelectedIndex])
}
@IBAction func onMyPicker(_ value: Int) {
       pikerSelectedIndex=value
       nameLabel.setText(namesArray[pikerSelectedIndex])
   }
```

```
@IBAction func onDetailButton() {
        self.presentController(withName: "detail",
                               context: pikerSelectedIndex)
}
```

3）宠物详情页面

在主页面单击详情按钮则弹出宠物详情页面 DetailInterfaceController，在详情页面初始化时确定要显示的是哪个宠物。详情页面主要包括图片、开关和滑动条等控件，开关控制图片是否显示，滑动条设置图片尺寸。

```
@IBOutlet var imageView: WKInterfaceImage! //图片
@IBOutlet var descriptionLabel: WKInterfaceLabel!  //描述文字
@IBOutlet var showSwitch: WKInterfaceSwitch!  //显示开关
@IBOutlet var withSlider: WKInterfaceSlider! //滑动条

override func awake(withContext context: Any?) {
    super.awake(withContext: context)
    // Configure interface objects here.
    let index=context as! Int
    descriptionLabel.setText(descriptionArray[index])
    imageView.setImageNamed(imageNameArray[index])
    let with:CGFloat=100
    withSlider.setWidth(with)
    imageView.setWidth(with)
    }
 //设置隐藏状态
@IBAction func changeSwitch(_ value: Bool) {
    imageView.setHidden(!value)
}
//设置使用滑动条设置图片大小
@IBAction func onSlider(_ value: Float) {
    imageView.setWidth(CGFloat(value))
}
```

4）表格显示

除了选择器，表格也可以列举显示所有宠物，这里我们在主页面并列建立一个表格页面 RightInterfaceController，通过表格控件 WKInterfaceTable 列出所有宠物，单击其中一行，则弹出相应的宠物详情页面。

与 iOS 系统的表格控件 UITableView 不同，手表表格控件 WKInterfaceTable 结构简单，默认每一行的控件是空的，所以设置表格的每行内容并做出相应动作，需要我们自己创建一个自定义的包含控件的行对象，并将其匹配到表格中，然后设置每行的内容。

（1）在 storyboard 里的页面上拖建一个表格控件 Table。

（2）将行的 Identifier 设为“myRowID”。

（3）在 WatchKit Extension 目录中新建文件，并新建一个行类 MyRowController，并在 storyboard 里将行类绑定为 MyRowController。

```
import Foundation
import WatchKit
class MyRowController: NSObject {
    var index=0 //行号
}
```

（4）在 storyboard 里的行里添加按钮控件。

（5）在行类 MyRowController 中建立按钮控件变量和方法。

```
    @IBOutlet var rowButton: WKInterfaceButton!
    @IBAction func onRowButton() {
    rightInterfaceController?.presentController(withName: "detail",
                                                context: index)
    //rightInterfaceController 是全局变量
      }
```

（6）在 RightInterfaceController 中设置表格的行数和每行的内容。

```
var  rightInterfaceController:WKInterfaceController?
class RightInterfaceController: WKInterfaceController {
       @IBOutlet var tableController: WKInterfaceTable!
```

```
   override func awake(withContext context: Any?) {
       super.awake(withContext: context)
       // Configure interface objects here.
       //把当前页面传给全局变量
       rightInterfaceController=self
       //设置表格内容
       setTable()
   }
   override func willActivate() {
       super.willActivate()
   }
```

```
    override func didDeactivate() {
        super.didDeactivate()
}
    //设置表格内容
    func setTable()
    {
        //设置表格行数和行的类型 id
        let rowCount=5
        tableController.setNumberOfRows(rowCount, withRowType: "myRowID")
        //设置每行内容
        for i in 0..<rowCount
        {
            let row=tableController.rowController(at: i) as! MyRowController
            row.rowButton.setTitle(namesArray[i])
            row.index=i
        }
    }
}
```

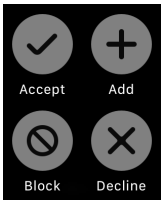
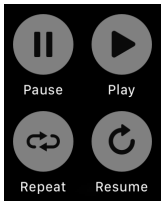
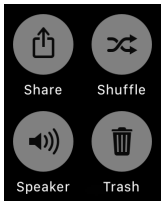
第 4 章

WatchOS 高级进阶

4.1 Force Touch 菜单

Apple Watch 具有著名的 Force Touch 按压技术，Force Touch 在应用中的具体表现形式就是按压应用界面弹出菜单。Xcode 提供了 16 种常用的菜单图标，如表 4-1 所示，同时我们也可以使用自己的图片，菜单图片图形是由透明通道 Alpha 决定的。

表 4-1　Force Touch 图标

中文	英文	菜单图标	中文	英文	菜单图标
接受	Accept		暂停	Pause	
添加	Add		播放	Play	
堵塞	Block		复位	Repeat	
下降	Decline		继续	Resume	
信息	Info		分享	Share	
疑问	Maybe		随机	Shuffle	
更多	More		外放	Speaker	
静音	Mute		回收	Trash	

4.2 振动引擎

手表具有振动功能，不静音时伴随与振动相同的铃声，系统提供了 9 种不同

的振动类型 WKHapticType，如表 4-2 所示。

```
//调用振动
WKInterfaceDevice.current().play(WKHapticType.notification)
```

表 4-2　振动类型

中文	英文	方式
通知	notification	强振，带有尾振
向上	directionUp	两次短振，升调
向下	directionDown	两次短振，降调
成功	success	三次短振，升调
失败	failure	三次短振，降调
重试	retry	三次短振，平调
开始	start	一次振动
停止	stop	两次振动，平调
点击	click	轻微振动动，无声音

4.3　表盘功能栏

4.3.1　功能栏简介

从 watchOS 2 开始，第三方应用通过功能栏 Complication 的形式在表盘上显示一些简单的图片和文字信息。功能栏 Complication 有多种类型，称为家族（Family），每种家族都与相应的表盘类型相匹配，并且有预设的显示格式模板（CLKComplicationTemplate），模板主要包括要显示的文字提供器（CLKTextProvider）和图片提供器（CLKImageProvider）。

功能栏的支持家族可以在项目 Extension 的 Complications Configuration 中勾选，功能栏的配置对应的默认类是 ComplicationController,ComplicationController，遵循功能栏数据源协议 CLKComplicationDataSource，默认会包括几个在不同时机触发的方法，其中当前时间显示内容的方法是 getCurrentTimelineEntry()，有好几个方法是用来配合时间旅行 TimeTravel 的，但是时间旅行只是 watchOS 2 的，在 watchOS 3 里面被取消了。

系统中功能栏服务器 CLKComplicationServer 负责功能栏的管理，当需要更

新表盘的功能栏显示内容时，我们要调用 CLKComplicationServer 完成刷新工作。

4.3.2 功能栏刷新

一般应用会在显示新信息的时候刷新表盘上该应用的功能栏，此时可调用表盘服务器刷新表盘上所有的功能栏。

```
/在程序任意处更新表盘
//刷新表盘功能栏 complication
let server = CLKComplicationServer.sharedInstance()
for complication in server.activeComplications!
{
    //刷新 complication 数据
    server.reloadTimeline(for: complication)
}
```

当表盘服务器刷新时，会调用 ComplicationController 的 getCurrentTimelineEntry() 函数，所以我们需要在该函数中设置要显示的功能栏。这里我们先只介绍模块表盘小功能栏的设置方法。

```
func getCurrentTimelineEntry(for complication: CLKComplication,
    withHandler handler: ((CLKComplicationTimelineEntry?) -> Void)) {
    let now=Date()
    var entry:CLKComplicationTimelineEntry?
    switch complication.family
    {
    case .modularSmall:
        //创建模版
        let
templateStackImage=CLKComplicationTemplateModularSmallStackImage()
        //设置模版的图片和文字
        let image=UIImage(named:"image")
        let text="wenzi"
        templateStackImage.line1ImageProvider=CLKImageProvider
(onePieceImage: image!)
```

```
        templateStackImage.line2TextProvider=CLKTextProvider.
        Localizable TextProvider(withStringsFileTextKey: text)
        //设置颜色
        templateStackImage.line1ImageProvider.tintColor=UIColor.
yellow
        //创建功能栏显示入口
        entry = CLKComplicationTimelineEntry(date: now,
                complicationTemplate: templateStackImage)
        default: break
handler(entry)
}
```

4.3.3　Watch 表盘图库示例

手机 Watch 应用里可以浏览和设置表盘，我们可以将自己应用的功能栏添加到此处供用户浏览和选择。要在手机 Watch 应用里添加功能栏示例，则在 ComplicationController 中编写 getLocalizableSampleTemplate()函数。

```
private func getLocalizableSampleTemplate(for complication:
CLKComplication,  withHandler handler: (CLKComplicationTemplate?)
-> Void)
{
         //  This  method  will  be  called  once  per  supported
complication, and the results will be cached
         let color0=UIColor.yellow
         switch complication.family
         {
         case .modularSmall:
         //创建模板
let templateSimpleText=CLKComplicationTemplateModularSmallSimpleText()
         //设置模板的文字
         templateSimpleText.textProvider=CLKTextProvider.
localizableTextProvider(withStringsFileTextKey: "  ")
```

```
            //设置颜色
                templateSimpleText.textProvider.tintColor=color0
                handler(templateSimpleText)
                default: break

        }
}
```

4.3.4 家族和模板

如上文所述，表盘类型、家族、模板都是相互匹配的，为了方便选择，这里列出所有的家族和模板，以及适用的表盘，如表 4-3 所示。

表 4-3 家族和模板

表盘类型	家族 family	模板 Template
Modular 模块	.modularSmall	CLKComplicationTemplateModularSmallSimpleText
		CLKComplicationTemplateModularSmallSimpleImage
		CLKComplicationTemplateModularSmallRingText
		CLKComplicationTemplateModularSmallRingImage
		CLKComplicationTemplateModularSmallStackImage
		CLKComplicationTemplateModularSmallStackText
	.modularLarge	CLKComplicationTemplateModularLargeTallBody
		CLKComplicationTemplateModularLargeColumns
		CLKComplicationTemplateModularLargeTable
		CLKComplicationTemplateModularLargeStandardBody
		CLKComplicationTemplateModularSmallColumnsText
Utilitarian 实用	.utilitarianSmall	CLKComplicationTemplateUtilitarianSmallFlat
		CLKComplicationTemplateUtilitarianSmallRingImage
		CLKComplicationTemplateUtilitarianSmallRingText
		CLKComplicationTemplateUtilitarianSmallSquare
	utilitarianSmallFlat	CLKComplicationTemplateUtilitarianSmallFlat
	.utilitarianLarge	CLKComplicationTemplateUtilitarianLargeFlat
Circular 简约 动态	.circularSmall	CLKComplicationTemplateCircularSmallRingImage
		CLKComplicationTemplateCircularSmallRingText
		CLKComplicationTemplateCircularSmallSimpleImage
		CLKComplicationTemplateCircularSmallSimpleText
		CLKComplicationTemplateCircularSmallStackImage
		CLKComplicationTemplateCircularSmallStackText
		CLKComplicationTemplateExtraLargeColumnsText

（续表）

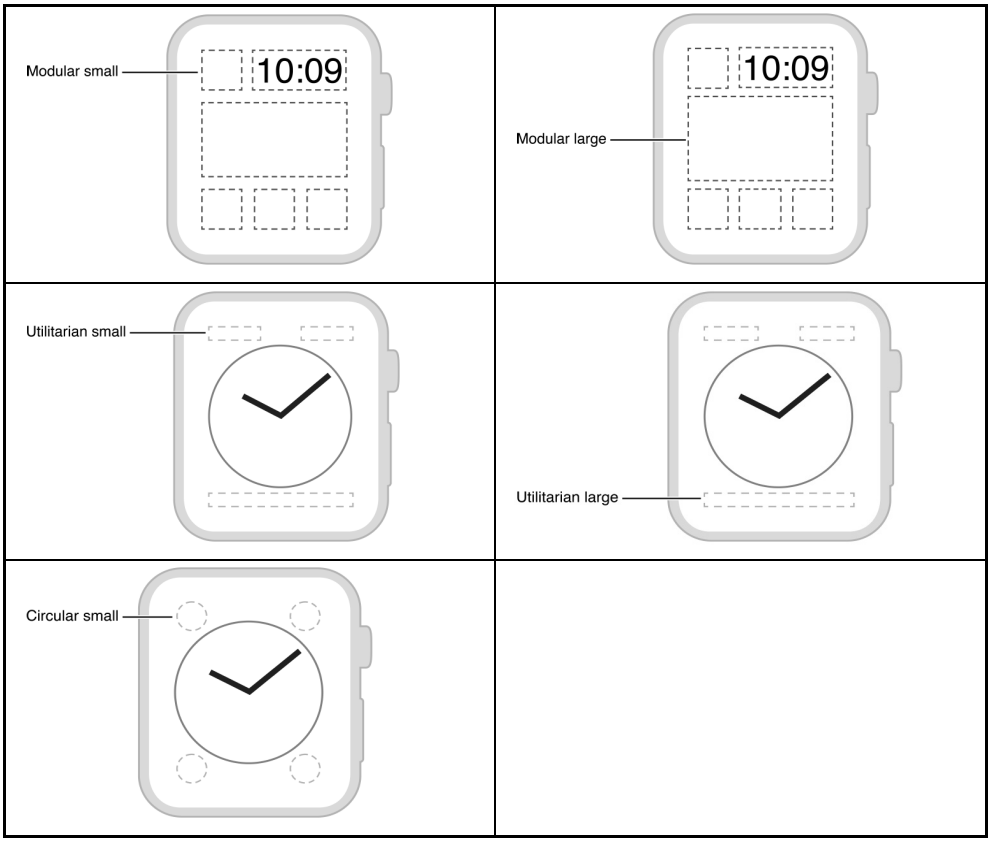
表盘类型	家族 family	模板 Template
ExtraLarge	.extraLarge	CLKComplicationTemplateExtraLargeRingImage
		CLKComplicationTemplateExtraLargeRingText
		CLKComplicationTemplateExtraLargeSimpleImage
		CLKComplicationTemplateExtraLargeSimpleText
		CLKComplicationTemplateExtraLargeStackImage
		CLKComplicationTemplateExtraLargeStackText

其中，CLKComplicationFamily.utilitarianSmallFlat 适用于 Utility、Mickey、Chronograph 和 Simple 表盘；CLKComplicationFamily.extraLarge 适用于 X-Large 表盘。

4.3.5　家族示意图

表 4-4 列出了功能栏的示意图（来自苹果官方开发文档）。

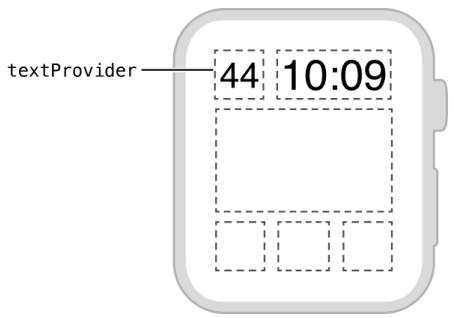
表 4-4　功能栏示意图表格

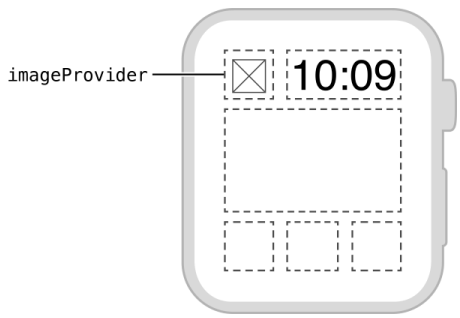


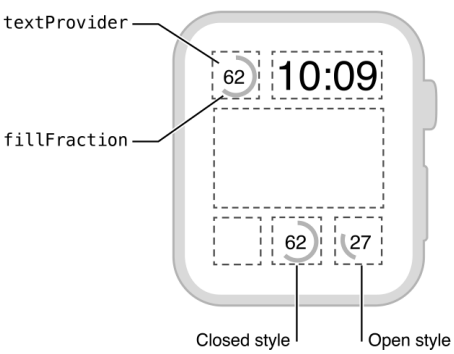
4.3.6 模板示意图

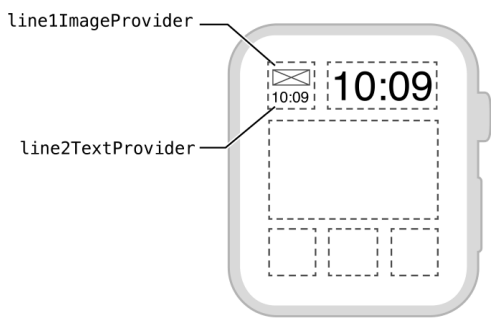
表 4-5 列出了所有的模板类型的示意图（来自苹果官方开发文档）。

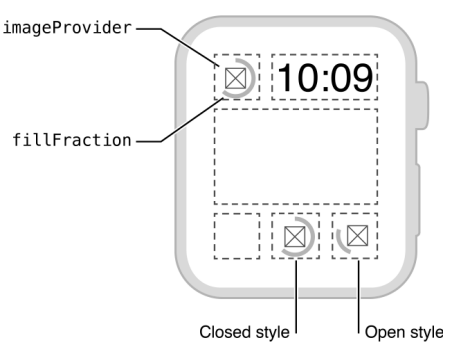
表 4-5 模板示意图表格

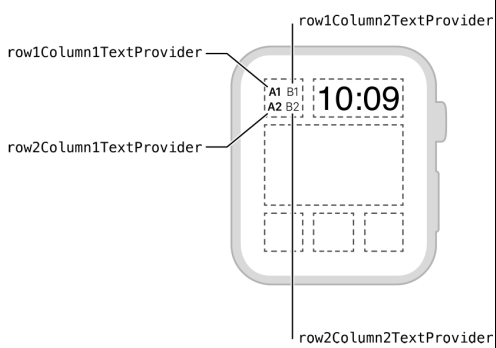
<table>
<tr>
<td> 

ModularSmallSimpleText</td>
<td> 

ModularSmallSimpleImage</td>
</tr>
<tr>
<td> 

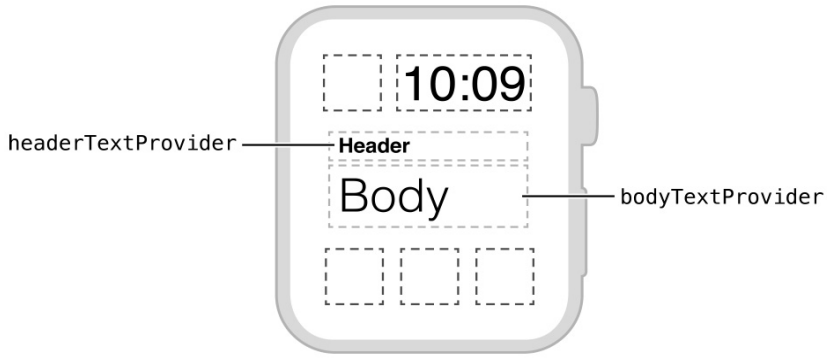
ModularSmallRingText</td>
<td> 

ModularSmallStackImage</td>
</tr>
<tr>
<td> 

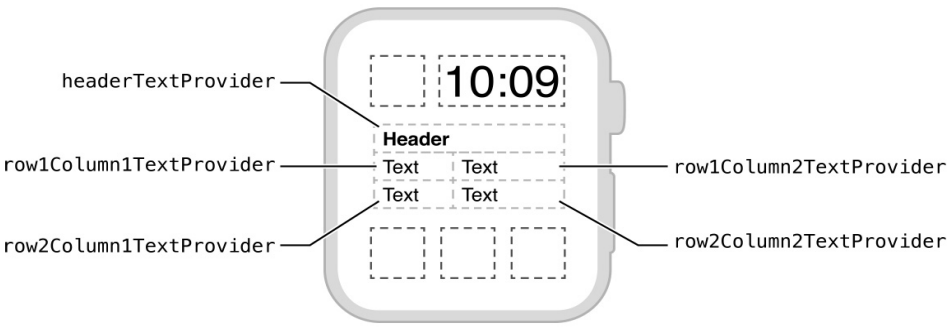
ModularSmallRingImage</td>
<td> 

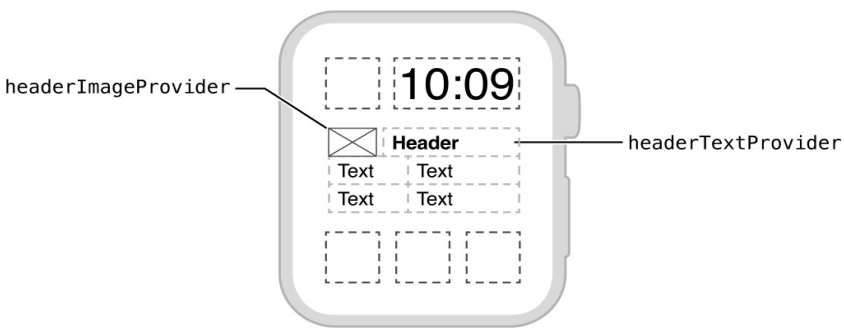
ModularSmallColumnsText</td>
</tr>
</table>

（续表）



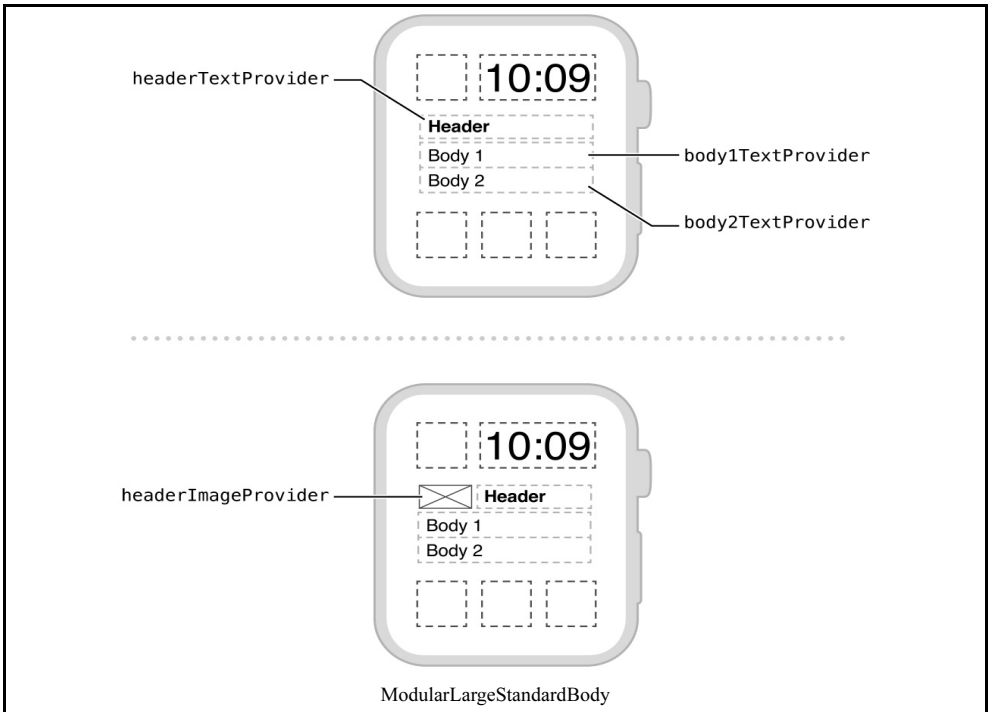
ModularLargeTallBody



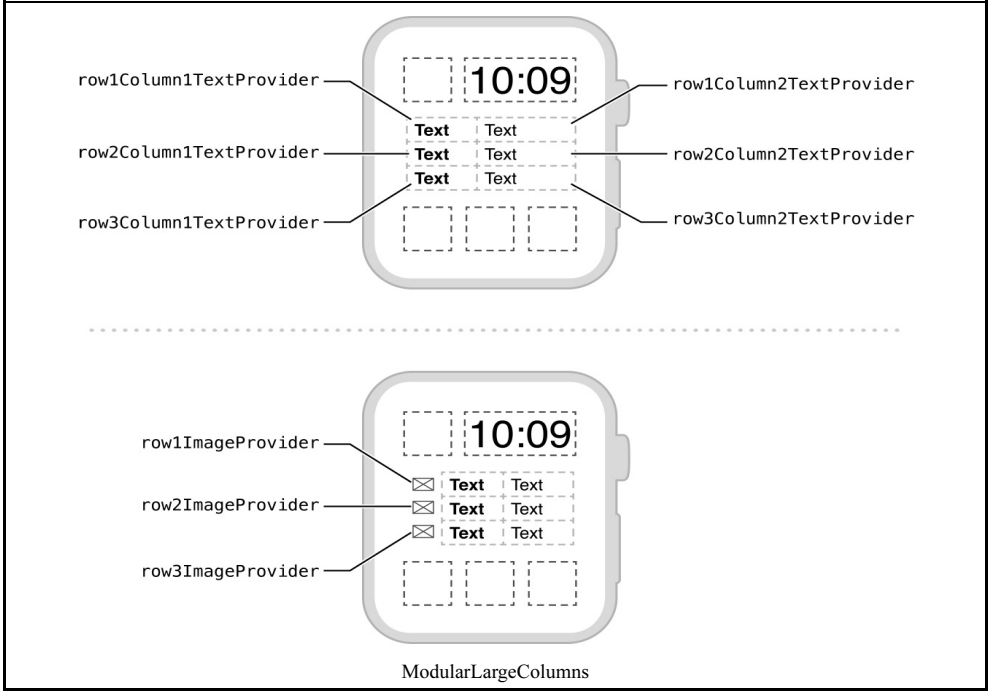


ModularLargeTable

（续表）

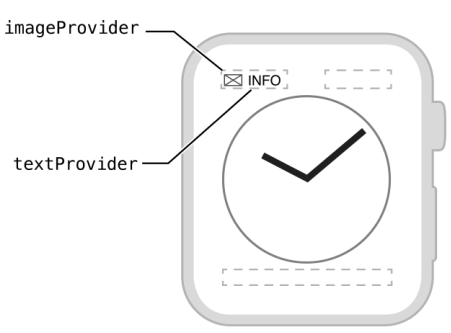


ModularLargeStandardBody

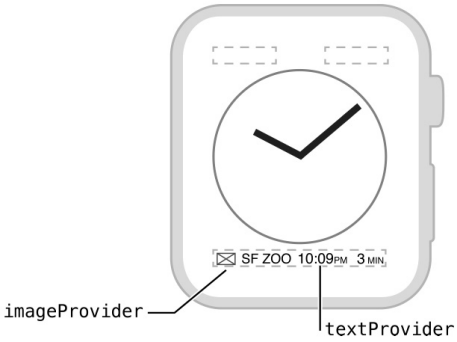


ModularLargeColumns

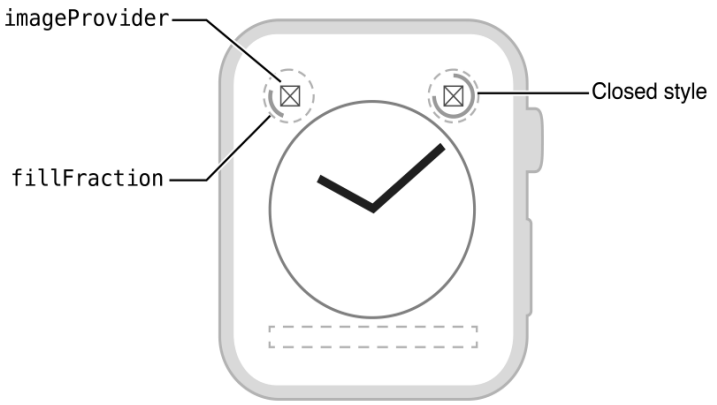
（续表）



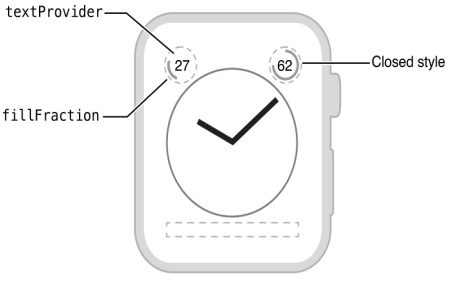
UtilitarianSmallFlat



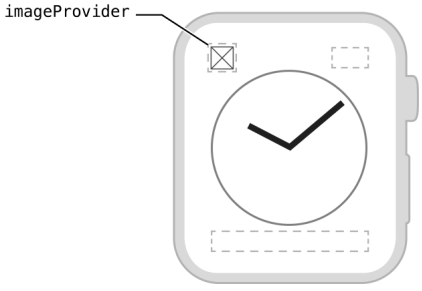
UtilitarianLargeFlat



UtilitarianSmallRingImage

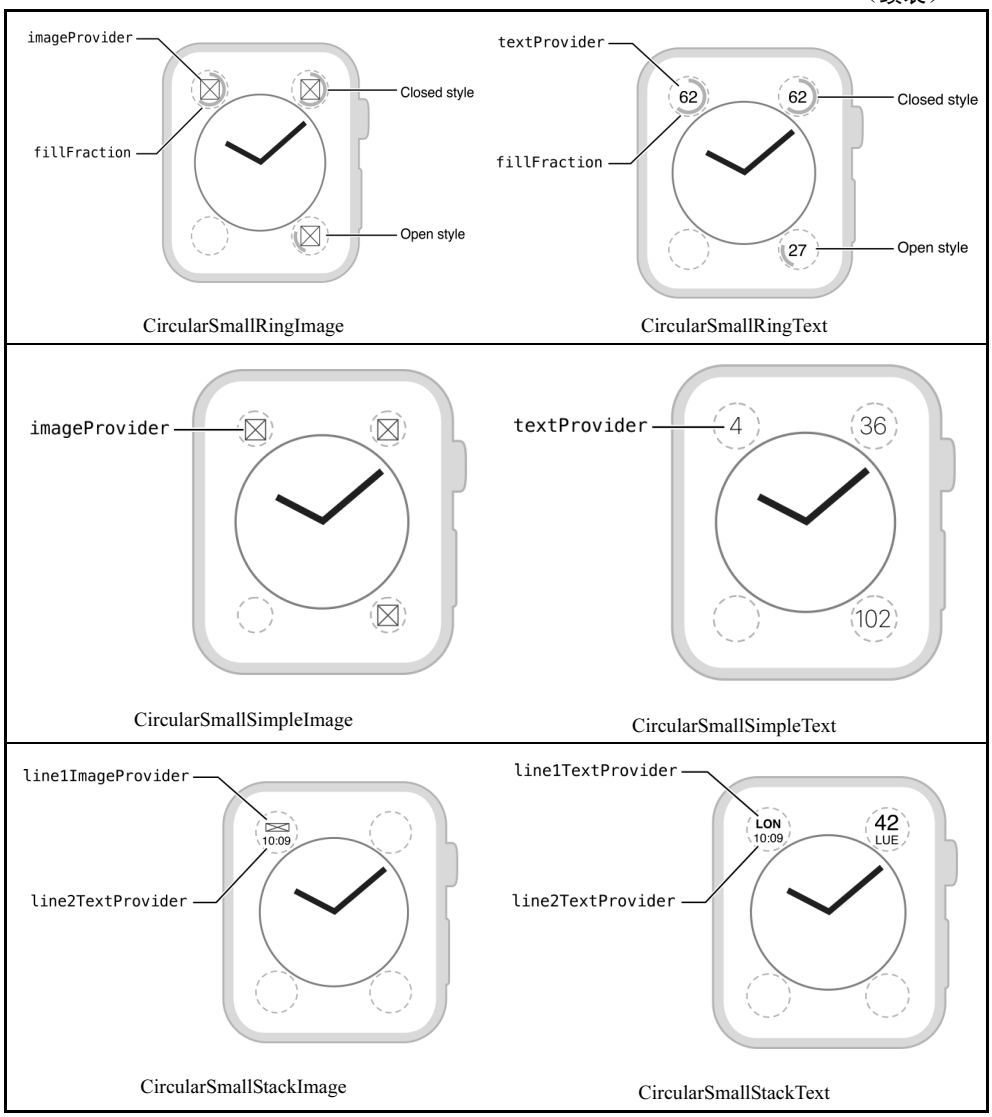


UtilitarianSmallRingText



UtilitarianSmallSquare

（续表）



4.3.7 功能栏图片尺寸

Apple watch 具有 38mm 和 42mm 两种款式，两种款式的屏幕大小是不同的，同时功能栏的模板类型不同，这都有可能会使显示图片大小不同。图片显示是@x2 模式。表 4-6 列出了各种模板的图片显示尺寸，如果图片文件本身不标准，系统则会进行自适应缩放。

表 4-6　功能栏图片尺寸

支持图片显示的模板	38mm/pixels Width/Height	42mm/pixels Width/Height
CLKComplicationTemplateModularSmallSimpleImage	52/52	58/58
CLKComplicationTemplateModularSmallRingImage	36/36	38/38
CLKComplicationTemplateModularSmallStackImage	52/28	58/30
CLKComplicationTemplateModularLargeColumns	22/22	24/24
CLKComplicationTemplateModularLargeTable	22/22	24/24
CLKComplicationTemplateModularLargeStandardBody	22/22	24/24
CLKComplicationTemplateUtilitarianSmallFlat	18/18	20/20
CLKComplicationTemplateUtilitarianSmallRingImage	28/28	28/28
CLKComplicationTemplateUtilitarianSmallSquare	40/40	44/44
CLKComplicationTemplateUtilitarianSmallFlat	18/18	20/20
CLKComplicationTemplateUtilitarianLargeFlat	18/18	20/20
CLKComplicationTemplateCircularSmallRingImage	40/40	44/44
CLKComplicationTemplateCircularSmallSimpleImage	32/32	36/36
CLKComplicationTemplateCircularSmallStackImage	32/14	34/16
CLKComplicationTemplateExtraLargeRingImage	108/108	114/114
CLKComplicationTemplateExtraLargeSimpleImage	182/182	203/203
CLKComplicationTemplateExtraLargeStackImage	156/84	174/90

4.4 提醒

在需要的时候可以弹出提醒框，提供一定的信息或者让用户做出选择。提醒框可以添加自定义的按钮，根据用户选择进行续后处理。

提醒框包括两部分：提醒信息和按钮。

按钮类型（WKAlertActionStyle）共有 3 种：default，白色文字按钮；cancel，左上角/默认；destructive，红色文字按钮。

提醒框本身分为 3 种类型，具体形式如表 4-7 所示。

表 4-7　提醒框类型

类型	alert	sideBySideButtonsAlert	actionSheet
描述	左上角没有“取消”	靠边按钮	左上角有“取消”

（续表）

类型	alert	sideBySideButtonsAlert	actionSheet
图示			

4.5 后台刷新任务

watchOS 3 提供了对后台刷新任务 WKApplicationRefreshBackgroundTask 的支持。后台刷新任务的运行不像其他操作系统常规意义的后台那样可以一直在后台保持一个线程的运行，而是在后台刷新任务的计划开启时间点之后，自动地每隔一个小时被系统调用一次。换句话说，系统会每隔一个小时自动遍历 dock 中的应用，如果应用挂启了后台任务，则会执行启动应用并执行应用的一个后台任务。一个应用只能挂启一个后台刷新任务，如果前一个后台刷新任务还未结束，挂启第二个会把前一个覆盖掉。

应用挂启后台刷新任务使用扩展 WKExtension 的方法 scheduleBackground Refresh(withPreferredDate:userInfo:scheduledCompletion:)，参数中 withPreferredDate 是执行最早的时间，系统会在该时间点后尝试调用后台；userInfo 是自定义的字典信息；scheduledCompletion 是回调函数，一般立刻执行，可以在里面检查挂启是否成功。

假如后台刷新任务被激发，系统会将应用唤醒，调用应用页面的激活方法 willActivate()，同时执行扩展代理 WKExtensionDelegate 中的处理方法 handle()来处理后台任务。我们可以检查扩展 WKExtension 的应用状态 applicationState 来判断应用是用户前台操作运行状态（active）还是后台调用状态（background）。后台刷新任务会周期性调用，所以需要我们手动结束 setTaskCompleted()。

4.6 URL 后台下载

watchOS 提供了对 URLSessionTask 的下载任务 downloadTask 的支持。提供一个下载地址 URL，构建一个 URLSession，然后调用方法 resume()就可以启动下载任务。

下载任务是后台模式 WKURLSessionRefreshBackgroundTask，watchOS 在下载开始时会调用 WKExtensionDelegate 中的处理方法 handle()，同时我们也可以使用 URL 下载代理 URLSessionDownloadDelegate 对整个下载过程进行检测。

```
//继续下载：
func urlSession(URLSession, downloadTask: URLSessionDownloadTask,
                didResumeAtOffset: Int64, expectedTotalBytes: Int64)
```

```
//下载进度，周期性调用
func urlSession(URLSession, downloadTask: URLSessionDownloadTask,
                 didWriteData: Int64,
             totalBytesWritten: Int64, totalBytesExpectedToWrite: Int64)

//下载结束
func urlSession(URLSession, downloadTask: URLSessionDownloadTask,
             didFinishDownloadingTo: URL)
```

4.7 Dock 截图

内存中的应用会保留在 Dock 中，其显示截图 Snapshots 默认是系统周期性地抓取应用运行时的显示内容。我们可以通过挂启 scheduleSnapshotRefresh (withPreferredDate:userInfo:scheduledCompletion:)后台截图刷新任务 WKSnapshot RefreshBackgroundTask 手动的抓取 Dock 截图。

```
//截屏，Dock 里的界面更新
func scheduleSnapshot() {
   // fire now, we're ready
   let fireDate = Date()

WKExtension.shared().scheduleSnapshotRefresh(withPreferredDate:
fireDate, userInfo: nil) { error in
           if (error == nil) {
              print("successfully scheduled snapshot.  All background
 work completed.")
           }
       }
}
```

4.8 Apple Pay 支付

wathOS3 提供一个支付按钮 WKInterfacePaymentButton，其上显示一个 Apple Pay 的特殊标志 Pay，与其他普通按钮一样，我们需要自行绑定一个执行函数进行支付流程，支付流程涉及一个支付授权页面 PKPaymentAuthorizationController，而具体支付流程比较复杂，不在本书涉及范围之内，请读者自行参考 Apple Pay 相关开发文档。

4.9 通知

当手机收到应用通知时，如果手机息屏且手表处于佩戴状态，手表就会收到通知推送，当然手表必须连接蓝牙或 WiFi。通知页面分静态和动态两种，可在 Xcode 的 storyboard 页面中见到，同时可在“NotificationController”的方法“didReceive()”中设置动态内容。

苹果新系统采用了新的通知框架 UserNotifications， UserNotifications 同时适用于 iOS 10 和 watchOS 3，但是 watchOS 3 对 UserNotifications 的支持还在逐步开发中。通知分为本地和远程，由于通知框架设计的内容比较繁多，且多使用于 iOS 端，不是手表开发的重点，所以本书不予详述。

4.10　【案例 3】十二生肖

本案例以一定的时间周期进行循环，每一次循环更新一个属相，同时进行振动和更新表盘功能栏，当循环十二次后进行提醒，用户可以继续循环和停止。本案例同时演示了振动引擎和 ForceTouch 菜单。

1. 项目文件结构

本项目主要涉及 3 个文件：“InterfaceController.swift”是十二生肖的主文件；“ComplicationController.swift”为表盘功能栏设置文件；“InterfaceController1.swift”为振动引擎演示文件。至于 Force Touch 菜单，只使用了界面演示。项目文件结构图如图 4-1 所示。

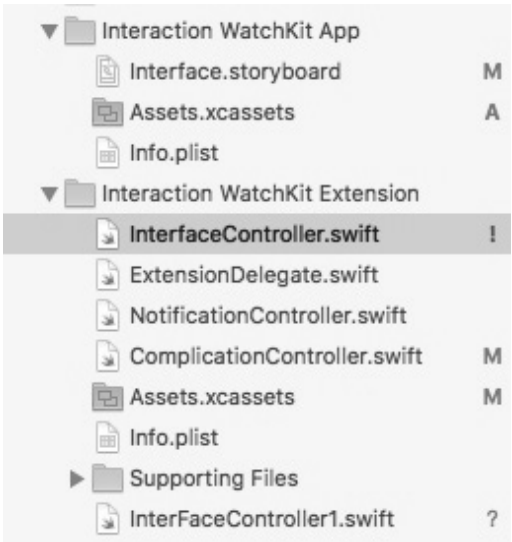


图 4-1　项目文件结构

2. 界面

“Interface Controller”是十二生肖的主页面；“Interface Controller1”为振动引擎演示页面。另外，Force Touch 菜单只使用了界面演示，包括 3 个页面，即“ForceTouch0”、“ForceTouch1”和“ForceTouch2”。界面设置如图 4-2 所示。

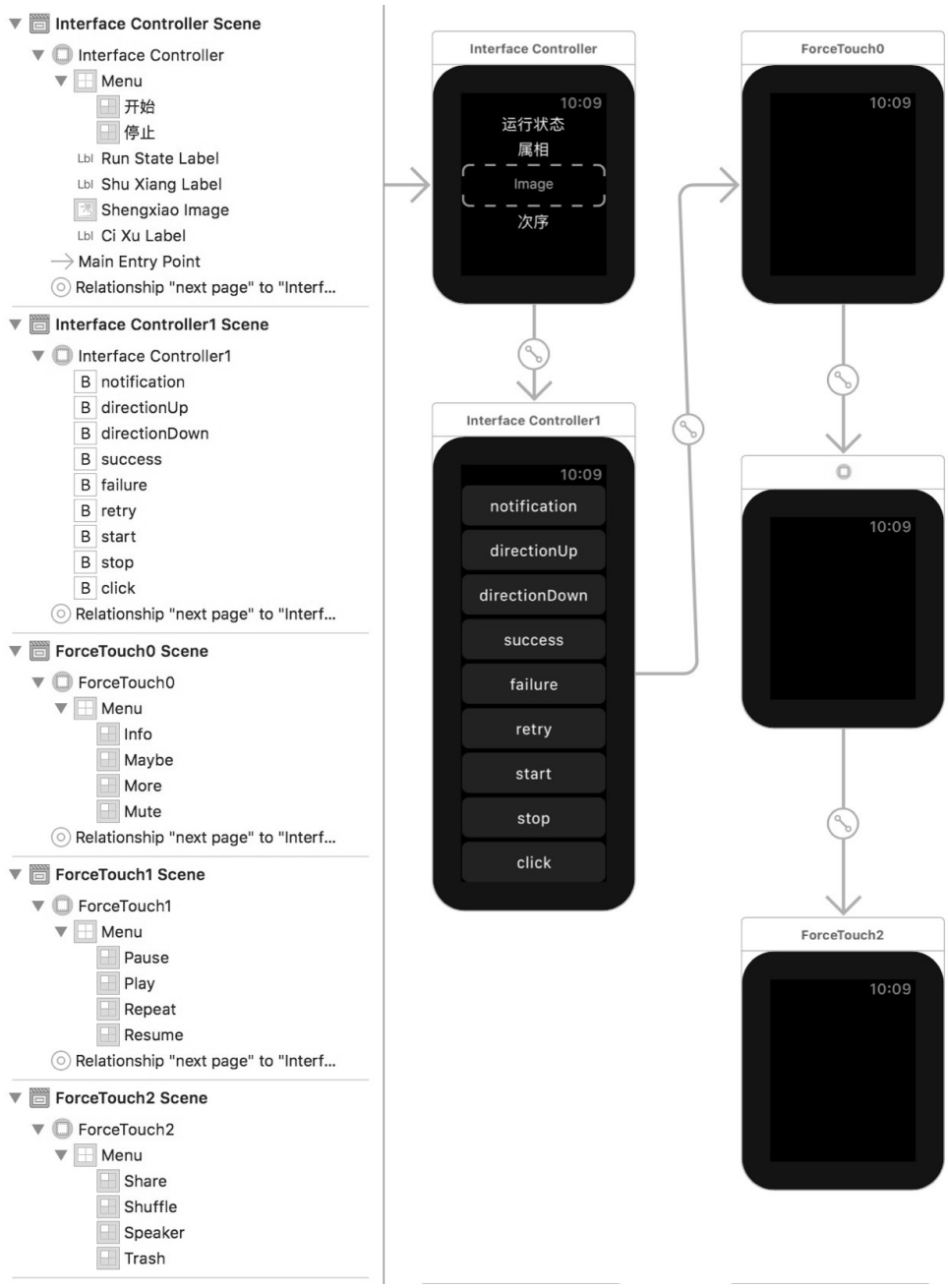

Interface Controller Scene
Interface Controller
Menu
开始
停止
Run State Label
Shu Xiang Label
Shengxiao Image
Ci Xu Label
Main Entry Point
Relationship "next page" to "Interf...
Interface Controller1 Scene
Interface Controller1
notification
directionUp
directionDown
success
failure
retry
start
stop
click
Relationship "next page" to "Interf...
ForceTouch0 Scene
ForceTouch0
Menu
Info
Maybe
More
Mute
Relationship "next page" to "Interf...
ForceTouch1 Scene
ForceTouch1
Menu
Pause
Play
Repeat
Resume
Relationship "next page" to "Interf...
ForceTouch2 Scene
ForceTouch2
Menu
Share
Shuffle
Speaker
Trash
Interface Controller
10:09
运行状态
属相
Image
次序
ForceTouch0
10:09
Interface Controller1
10:09
notification
directionUp
directionDown
success
failure
retry
start
stop
click
10:09
ForceTouch2
10:09

图 4-2　界面设置

3. 开发流程

1）十二生肖资料

主页面和表盘都要显示相同的十二生肖，所以先建立全局的十二生肖资料，以备调用。

```
//全局
var shiErShengXiao=[("子　鼠","shu"),("丑　牛","niu"),("寅　虎","hu"),
                    ("卯　兔","tu"),("辰　龙","long"), ("巳　蛇","she"),
                    ("午　马","ma"),("未　羊","yang"),("申　猴","hou"),
                    ("酉　鸡","ji"),("戌　狗","gou"),("亥　猪","zhu")]
```

2）建立计时器

本案例以一定的时间周期进行循环，这里使用计时器 Timer 来实现，需要注意的是，Timer 对象不能是临时变量，而是要建立成页面的成员变量。在开始按钮方法里进行循环初始化，然后建立 Timer 对象并开启 3 秒循环迭代。

```
    //计时器
    var timer:Timer?
    //属相序列
    var shuxiang=0

@IBAction func onStartMunu() {
        //初始化
        shuxiang=0
        //跟新页面和表盘
        update()
        //开启计时器，3 秒钟重复循环
        timer=Timer.scheduledTimer(timeInterval: 3, target: self,
selector: #selector(InterfaceController.updateTimer), userInfo:
nil, repeats:true)
        //timer?.fire()//不 fire 则 3 秒之后触发

        runStateLabel.setText("循环中")
    }
```

3）计时器迭方法

计时器开始循环后，每 3 秒会执行一次循环函数“updateTimer”，我们在该函数中进行属相更换，使用新的属相进行刷新页面和表盘。每次执行时都要判断是否为 12 次，当 12 次后进行提醒。

```
    func updateTimer(uTimer:Timer)
        {
            //循环
            shuxiang+=1
            //循环一轮则提示
            if shuxiang==12
            {
                onStopMenu()
                alert()
            }else
            {
                shuxiang%=12
                update()
            }
        }
```

4）循环提醒

提醒另做函数 alert()。把某一块代码打包成一个函数调用，使得代码简洁、整齐、易读，这是一个好习惯。

```
    //提醒
    func alert()
    {
        //设置按钮
           let  alertAction0=WKAlertAction(title:  " 继 续 ",  style:
WKAlertActionStyle.default)
        {()->Void in
            self.onStartMunu()//重新开始
        }
        let alertAction1=WKAlertAction(title: "停止", style:
WKAlertActionStyle.default)
```

```
        {()->Void in
            self.onStopMenu()//结束循环
        }
        //弹出提醒页面
      presentAlert(withTitle: "循环完毕", message: "是否继续循环",
preferredStyle: WKAlertControllerStyle.actionSheet, actions:
[alertAction0,alertAction1])
   }
```

5）页面和表盘更新

每次循环都更新页面和表盘，我们打包在 update()函数中。

```
    //跟新页面和表盘
    func update()
    {
        //显示在表盘
        shuXiangLabel.setText(shiErShengXiao[shuxiang].0)
        ciXuLabel.setText(String(shuxiang))
        //let image=UIImage(named: shiErShengXiao[shuxiang].1)
        shengxiaoImage.setImageNamed(shiErShengXiao[shuxiang].1)

        //存储在用户默认参数中,供表盘更新使用
        let userDefaults=UserDefaults.standard
        userDefaults.set(shuxiang, forKey: "shuxiang")
        userDefaults.synchronize()

        //振动反馈
        WKInterfaceDevice.current().play(WKHapticType.notification)
```

```
     //刷新表盘功能栏 complication
        let server = CLKComplicationServer.sharedInstance()
        for complication in server.activeComplications!
        {
            //刷新 complication 数据
            server.reloadTimeline(for: complication)
        }
    }
```

6）设置表盘

功能栏服务器更新表盘会调用 ComplicationController 的 getCurrentTimeline Entry()函数。

```
   func getCurrentTimelineEntry(for complication: CLKComplication,
withHandler handler: @escaping ((CLKComplicationTimelineEntry?) ->
Void))
    {
        // Call the handler with the current timeline entry

        let now=Date()
        var entry:CLKComplicationTimelineEntry?
        let userDefaults=UserDefaults.standard
        let shuxiang=userDefaults.integer(forKey: "shuxiang")
        let image=UIImage(named: shiErShengXiao[shuxiang].1)
        let text=shiErShengXiao[shuxiang].0
        let color0=UIColor.yellow

        //按照表盘类型和家族设置模板
        //表盘类型、家族和模板对应关系见表 4-3
```

```
        switch complication.family
        {
        case .modularSmall:
            //创建模板
         let
templateStackImage=CLKComplicationTemplateModularSmallStackImage()
            //设置模板的图片和文字
             templateStackImage.line1ImageProvider=CLKImageProvider
(onePieceImage: image!)
            templateStackImage.line2TextProvider=CLKTextProvider.
localizableTextProvider(withStringsFileTextKey: text)
            //设置颜色
            templateStackImage.line1ImageProvider.tintColor=color0
            //创建功能栏显示入口
```

```
            entry = CLKComplicationTimelineEntry(date: now,
complicationTemplate: templateStackImage)

        case .modularLarge:
            let
templateLarger=CLKComplicationTemplateModularLargeStandardBody()
            templateLarger.headerImageProvider=CLKImageProvider(on
ePieceImage: image!)
            templateLarger.headerTextProvider=CLKTextProvider.loca
lizableTextProvider(withStringsFileTextKey: "生肖")
            templateLarger.body1TextProvider=CLKTextProvider.local
izableTextProvider(withStringsFileTextKey: text)
            templateLarger.body2TextProvider=CLKTextProvider.local
izableTextProvider(withStringsFileTextKey: String(shuxiang))
            templateLarger.headerImageProvider?.tintColor=color0
            templateLarger.headerTextProvider.tintColor=color0
```

```
            //templateLarger.body1TextProvider.tintColor=color1
            //templateLarger.body2TextProvider?.tintColor=color1

            entry = CLKComplicationTimelineEntry(date: now,
complicationTemplate: templateLarger)

        case .utilitarianSmall:
            let
templateRingImage=CLKComplicationTemplateUtilitarianSmallRingImage()
            templateRingImage.imageProvider=CLKImageProvider(onePi
eceImage: image!)
            templateRingImage.ringStyle=CLKComplicationRingStyle.open
            templateRingImage.imageProvider.tintColor=color0
            entry = CLKComplicationTimelineEntry(date: now,
complicationTemplate: templateRingImage)
```

```
        case .utilitarianSmallFlat:break

        case .utilitarianLarge:
            let
templateLarger=CLKComplicationTemplateUtilitarianLargeFlat()
            templateLarger.imageProvider=CLKImageProvider(onePiece
Image: image!)
            templateLarger.textProvider=CLKTextProvider.localizabl
eTextProvider(withStringsFileTextKey: text)
            templateLarger.imageProvider?.tintColor=color0
            //templateLarger.textProvider.tintColor=color1//无效
            entry = CLKComplicationTimelineEntry(date: now,
complicationTemplate: templateLarger)
```

```
        case .circularSmall:
            let
templateSimpleImage=CLKComplicationTemplateCircularSmallSimpleImage()
            templateSimpleImage.imageProvider=CLKImageProvider(one
PieceImage: image!)

            templateSimpleImage.imageProvider.tintColor=color0

            entry = CLKComplicationTimelineEntry(date: now,
complicationTemplate: templateSimpleImage)

        case .extraLarge:break

        //default: break
        }
        handler(entry)
    }
```

7）Watch 表盘图库示例

设置 iPhone Watch 应用中的表盘图库示例，需要充填 getLocalizableSampleTemplate()函数。

```
    // MARK: - Placeholder Templates
   private func getLocalizableSampleTemplate(for complication:
CLKComplication, withHandler handler: (CLKComplicationTemplate?) -> Void)
{
        // This method will be called once per supported complication,
and the results will be cached
        let color0=UIColor.yellow
```

```
       switch complication.family
       {
       case .modularSmall:
       //创建模板
       let
templateSimpleText=CLKComplicationTemplateModularSmallSimpleText()
       //设置模板的文字
       templateSimpleText.textProvider=CLKTextProvider.localizable
TextProvider(withStringsFileTextKey: "  ")
       //设置颜色
          templateSimpleText.textProvider.tintColor=color0
          handler(templateSimpleText)
          break
          default: break
        }
   }
```

8）振动引擎演示

Apple Wath 提供了多种振动，为了方便选择使用，这里在“InterFaceController1”页面进行所有振动类型的演示。

```
@IBAction func onButton0() {
       WKInterfaceDevice.current().play(WKHapticType.notification)
   }
   @IBAction func onButton1() {
       WKInterfaceDevice.current().play(WKHapticType.directionUp)
   }
   @IBAction func onButton2() {
       WKInterfaceDevice.current().play(WKHapticType.directionDown)
   }
```

```
@IBAction func onButton3() {
    WKInterfaceDevice.current().play(WKHapticType.success)
}
@IBAction func onButton4() {
    WKInterfaceDevice.current().play(WKHapticType.failure)
}
@IBAction func onButton5() {
    WKInterfaceDevice.current().play(WKHapticType.retry)
}
@IBAction func onButton6() {
    WKInterfaceDevice.current().play(WKHapticType.start)
}
@IBAction func onButton7() {
    WKInterfaceDevice.current().play(WKHapticType.stop)
}
@IBAction func onButton8() {
    WKInterfaceDevice.current().play(WKHapticType.click)
}
```

9）Force Touch 菜单演示

建立 3 个页面，每个页面添加 3 个 Force Touch 菜单，将表 4-1 的 12 个默认图标设置在 3 个菜单上，如图 4-2 所示。不必建立页面类，在运行时按压对应页面就会弹出设置的菜单。

4.11 【案例 4】后台刷新任务和 URL 下载

本案例是苹果官方案例，展示了后台刷新任务和 URL 下载的使用方法，笔者在官方源代码的基础上添加了补充注释，程序的基本运行流程如下。

（1）选定一个时间点，挂启后台刷新任务。

（2）在上述时间点后系统尝试调用应用，执行后台代理方法。

（3）在后台代理方法中激活 URL 后台下载任务。

（4）结束后台刷新任务。

（5）URL 后台下载任务下载文件，下载过程中可以监控进度。

（6）刷新 Dock 应用截图。

1）代码文件和界面

本案例只有一个界面，所以只涉及一个主页面文件“MainInterfaceController.swift”，界面包含一个时间文字标签和一个后台刷新任务开启按钮，如图 4-3 所示。

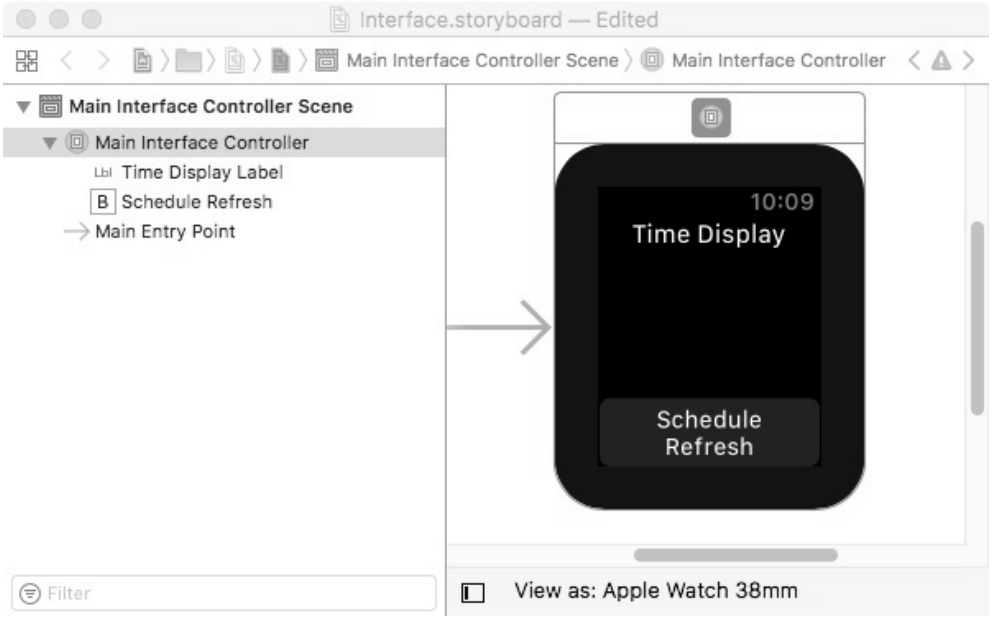


图 4-3　界面设置

2）开发流程

（1）挂启一个 20s 后的后台刷新任务（20s 后系统周期性地尝试启动本应用，但不一定会执行）。

```
//按钮
    @IBAction func ScheduleRefreshButtonTapped() {
        // fire in 20 seconds
        //20s 后激发
        let fireDate = Date(timeIntervalSinceNow: 20.0)
        // optional, any SecureCoding compliant data can be passed here
        //选用，放入自定义的信息
        let userInfo = ["reason" : "background update"]
```

```
    WKExtension.shared().scheduleBackgroundRefresh(withPreferredD
ate: fireDate, userInfo: userInfo as NSSecureCoding?) { (error) in
          if (error == nil) {
          //成功挂启后台刷新任务
             print("successfully scheduled background task, use the
crown to send the app to the background and wait for handle:BackgroundTasks
to fire.")
            }
         }
     }
```

（2）系统后台调用本应用执行后台任务，后台任务方法是扩展代理 WKExtensionDelegate 的接口。本案例在主页面里执行后台刷新任务，所以使主页面继承扩展代理 WKExtensionDelegate，并重写（override）后台方法 handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>)。

（3）上述 handle()方法会处理多种后台任务，当后台类型是 WKApplication RefreshBackgroundTask 时我们启动 URL 后台下载任务 scheduleURLSession()，然后使用 task.setTaskCompleted()关闭第（1）步开始的周期性后台刷新任务。

```
override func awake(withContext context: Any?) {
       super.awake(withContext: context)
       // Configure interface objects here.
     //设置扩展代理
       WKExtension.shared().delegate = self
   }
   //扩展代理
   func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
      for task : WKRefreshBackgroundTask in backgroundTasks {
         print("received background task: ", task)
```

```
      //只有是后台状态才处理
      if (WKExtension.shared().applicationState == .background) {
              if task is WKApplicationRefreshBackgroundTask {
              //此任务完后，应用将挂启，后台下载开始运行
                print("application task received, start URL session")
                scheduleURLSession()
              }
        }
else if let urlTask = task as? WKURLSessionRefreshBackgroundTask {
            let backgroundConfigObject =
URLSessionConfiguration.background(withIdentifier:
urlTask.sessionIdentifier)
              let backgroundSession = URLSession(configuration:
backgroundConfigObject, delegate: self, delegateQueue: nil)
              print("Rejoining session ", backgroundSession)
        }
        //如果不处理，要确定结束所有任务
        task.setTaskCompleted()
     }
   }
```

（4）URL 后台下载任务开始下载文件。

```
   func scheduleURLSession() {
       let backgroundConfigObject =
URLSessionConfiguration.background(withIdentifier: NSUUID().uuidString)
       backgroundConfigObject.sessionSendsLaunchEvents = true
       let backgroundSession = URLSession(configuration:
backgroundConfigObject)
       let downloadTask = backgroundSession.downloadTask(with:
sampleDownloadURL)
       downloadTask.resume()
   }
```

（5）URL 下载可以监控，同时随下载进度刷新页面和截图。

```
    //URLSession 处理
        func urlSession(_ session: URLSession, downloadTask:
URLSessionDownloadTask, didFinishDownloadingTo location: URL) {
            print("NSURLSession finished to url: ", location)
            updateDateLabel()
            scheduleSnapshot()
        }
```

（6）watchOS 3 支持对应用截图的刷新，即在 Dock 中显示应用截图。

```
//截屏，Dock 里的界面更新
    func scheduleSnapshot() {
        // fire now, we're ready
        let fireDate = Date()

WKExtension.shared().scheduleSnapshotRefresh(withPreferredDate:
fireDate, userInfo: nil) { error in
                if (error == nil) {
                    print("successfully scheduled snapshot. All background
work completed.")
                }
            }
```

第 5 章

多媒体和游戏引擎

5.1 多媒体

从 2.0 开始，watchOS 开放了多媒体 API，包括前台录音、无线播放音频、视频播放器，其中视频播放器使用喇叭外放声音。

5.1.1 录音

watchOS 2.0 开放了一个录音界面，提供前台录音功能。录音文件的扩展名只能取.wav、.mp4 和 .m4a，若设置成其他的会报错。录音文件要保存在组目录中。

```
self.presentAudioRecorderController(withOutputURL: fileURL!,
                preset: WKAudioRecorderPreset.highQualityAudio,
                options: nil)
   {(isSaved,error) in
          //print(error)
          if isSaved
          {
              //保存后续
          }
      }
   }
```

5.1.2 无线播放音频

watchOS 2.0 支持无线蓝牙耳机，提供了专用的无线音频播放器

WKAudioFilePlayer，可以播放单个音频文件，也可以播放音频序列，采取无线蓝牙播放时系统会提示 AirPlay 连接确认。音频播放器 WKAudioFilePlayer 的构建顺序为音频文件地址 URL、音频资源 WKAudioFileAsset、播放项目/播放序列、播放器 WKAudioFilePlayer。

```
let audioasset=WKAudioFileAsset(url: fileURL!)
let audioplyeritem=WKAudioFilePlayerItem(asset: audioasset)
let audiofileplayer=WKAudioFilePlayer(playerItem: audioplyeritem)
audiofileplayer.play()
```

5.1.3 视频播放和喇叭外放

watchOS 2.0 提供一个播放界面，支持播放音频和视频，声音是通过喇叭外放的。视频播放器不支持在线视频。

```
let mainbundle=Bundle.main
let videoURL=mainbundle.url(forResource: "IMG_1433", withExtension: "mp4")
self.presentMediaPlayerController(with: videoURL!, options: nil)
     {(bool,timeInterval,error) in
     //结束回调
          }
```

5.2 游戏引擎

watchOS 3.0 为游戏开发提供了两个控件，即 WKInterfaceSKScene 和 WKInterfaceSCNScene，其中控件 WKInterfaceSKScene 支持 2D 引擎 SpriteKit，控件 WKInterfaceSCNScene 支持 3D 引擎 SceneKit。

5.2.1 2D 游戏引擎控件

控件 WKInterfaceSKScene 支持 2D 引擎 SpriteKit。SpriteKit 一般包括一个基本 2D 场景 sks 文件，我们可以在 Interface.storyboard 中指定控件 WKInterfaceSKScene 的场景文件 sks，也可以在代码中构建一个 SKScene 实例，

然后使用函数 presentScene(_ scene: SKScene?)来显示游戏场景。

5.2.2　创建手表游戏项目

直接新建项目 Project，在打开的 watchOS 项目创建对话框中只能看到一项“iOS App with WatchKit App”，而没有 watchOS 的专属游戏项目。若要创建默认的手表游戏项目，需要先创建 iOS 项目，在 iOS 项目创建完成后，使用菜单新建 Target，就会在 watchOS 项目对话框中看到游戏项目“Game App”。

5.2.3　3D 游戏引擎控件

控件 WKInterfaceSCNScene 支持 3D 引擎 SceneKit。SceneKit 一般包括一个基本 3d 场景 scn 文件，我们可以在 Interface.storyboard 中指定控件 WKInterfaceSCNScene 的场景文件 scn（见表 5-1），也可以在代码中构建一个 SCNScene 实例，然后将该实例赋给 WKInterfaceSCNScene 的属性 scene。

控件 WKInterfaceSCNScene 也同时支持 2D 引擎 SpriteKit，在代码中构建一个 SKScene 实例，然后将该 2D 实例赋值给 WKInterfaceSCNScene 的属性 overlaySKScene 即可。

表 5-1　3D 游戏引擎控件配置

配置	中文描述
Scene	场景文件 scn
Antialiasing	抗锯齿
Frame Rate	刷新率
Default Lighting	默认灯光
Jitter	渲染抖动
Play	动画播放
Loop	动画循环

5.2.4　手势识别

watchOS 3.0 提供了 4 种手势识别器，可将其绑定到游戏窗口上实现人机交互。

（1）单击手势识别器（WKTapGestureRecognizer）识别单击手势，产生单击位置点 locationInObject()。

（2）扫动手势识别器（WKSwipeGestureRecognizer）识别扫动手势，产生上、

下、左、右四个扫动方向（WKSwipeGestureRecognizerDirection）。

（3）长按手势识别器（WKLongPressGestureRecognizer）识别长按手势，产生长按位置点 locationInObject()。

（4）拖动手势识别器（WKPanGestureRecognizer）识别拖动手势，产生拖动位置点 locationInObject()，此时需要注意手势的状态（state），如开始、变化、结束、取消等，在不同的状态进行不同的操作。

上述 4 种手势识别器具有共同的父类 WKGestureRecognizer，WKGestureRecognizer 常用的属性为位置（locationInObject()）和状态（state: WKGestureRecognizerState）。

5.3 【案例 5】录音和音频视频播放

本案例主要演示录音、耳机播放音频、喇叭外放、视频播放。

1. 文件和界面

本案例只含有一个页面，对应文件为“InterfaceController.swift”，页面上有 4 个按钮：录音、耳机、外放、视频播放示例，如图 5-1 所示。

图 5-1　界面设置

2. 开发流程

我们依次进行演示录音、耳机播放音频、喇叭外放音频和视频播放，视频播放时使用喇叭外放声音。

（1）按动“录音”按钮进行录音，录音完成将音频保存在组目录中，记录保存地址，以供以后播放。

```
var fileURL:URL? //录音文件地址
override func awake(withContext context: Any?) {
        super.awake(withContext: context)
        //group 地址
        let groupURL =
FileManager.default.containerURL(forSecurityApplicationGroupIdent
ifier: "group.tinghe17.AudioVideo")
        //录音文件地址
        fileURL=groupURL?.appendingPathComponent("recordFile.mp4")
    }
@IBAction func onRecord() {
        let recordOptions =
        [WKAudioRecorderControllerOptionsAlwaysShowActionTitleKey:true,
        WKAudioRecorderControllerOptionsAutorecordKey:true]
        self.presentAudioRecorderController(withOutputURL: fileURL!,
preset: WKAudioRecorderPreset.highQualityAudio, options: recordOptions)
        {(isSaved,error) in
            print(error as Any)
            if isSaved
            {
            }
        }
}
```

（2）录音后进行耳机播放，如果没有连接耳机，则系统会提醒。播放时依次按照音频文件 URL、WKAudioFileAsset、WKAudioFilePlayerItem、WKAudioFilePlayer 的顺序进行构建。

```
@IBAction func onAirPlay() {
   if fileURL==nil {return}
   let audioAsset=WKAudioFileAsset(url: fileURL!)
   let audioPlayerItem=WKAudioFilePlayerItem(asset: audioAsset)
   let audioFilePlayer=WKAudioFilePlayer(playerItem: audioPlayerItem)
   audioFilePlayer.play()
}
```

（3）以喇叭外放的形式播放音频文件，这里使用视频播放器实现。

```
 @IBAction func onSpeaker() {
        self.presentMediaPlayerController(with: fileURL!, options: nil)
        {(bool,timeInterval,error) in
        }
    }
```

（4）接下来进行视频播放。先将视频文件“IMG_1433.mp4”添加到扩展包 WatchKit Extension 中，然后就可以在代码中使用视频播放页面播放该视频了。

```
@IBAction func onPlayVideo() {
        let mainbundle=Bundle.main
        let videoURL=mainbundle.url(forResource: "IMG_1433",
withExtension: "mp4")
        self.presentMediaPlayerController(with: videoURL!, options: nil)
        {(bool,timeInterval,error) in
        }
}
```

5.4 【案例 6】2D 游戏

本案例展示 2D 游戏引擎控件 WKInterfaceSKScene 的使用方法，且可以通过手势识别器进行人机交互，手指操作屏幕可以以多种方式改变闪烁光点，以及文字标签的位置。

1. 创建项目

先创建 iOS 项目，在 iOS 项目创建完成后，使用菜单新建 Target，就会在

watchOS 项目对话框中看到游戏项目“Game App”，选择“Game App”并创建。

2. 项目文件目录

本项目主要涉及 3 个文件：主页面文件“InterfaceController.swift”、2D 游戏引擎 sks 文件“GameScene.sks”、2D 引擎对应的控制文件“GameScene.swift”。Watch Extension 目录结构如图 5-2 所示。

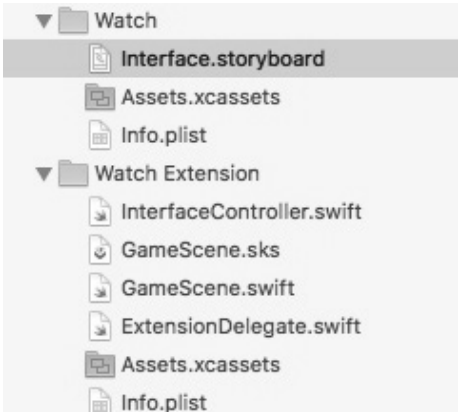


图 5-2　Watch Extension 文件目录

3. 界面设置

本项目只包含一个页面，如图 5-3 所示，其上包含一个全屏的 2D 游戏引擎控件“SprikeKit Scene”，我们在“SprikeKit Scene”控件上设置 4 种手势识别器：单击识别“Tap Gesture Recognizer”、扫动识别“Swipe Gesture Recognizer”、长按识别“Long Press Gestrue Recognizer”和拖动识别“Pan Gesture Recognizer”。

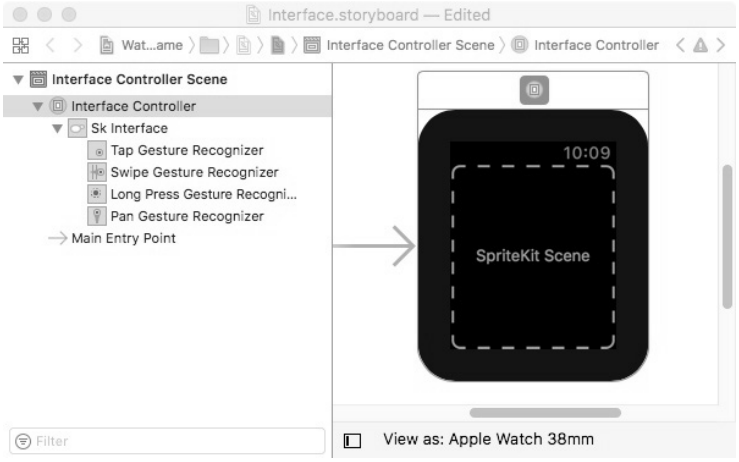


图 5-3　界面设置

4. 开发流程

（1）“Game App”创建后会默认包含游戏场景对应的控制代码，使用 GameScene 类来代表游戏场景，GameScene 类的成员和设置都与游戏场景想匹配，其中 spinnyNode 代表闪烁的光点，我们将其参数稍加修改，使其以更高频率闪烁；helloLabel 是 2D 文字标签。

```
import SpriteKit
class GameScene: SKScene {
    var spinnyNode : SKShapeNode?
    var helloLabel:SKLabelNode?
    override func sceneDidLoad() {
    helloLabel = self.childNode(withName: "//helloLabel") as? SKLabelNode
    helloLabel?.alpha = 0.0
    helloLabel?.run(SKAction.fadeIn(withDuration: 2.0))
    let w = (self.size.width + self.size.height) * 0.05
    let spinnyNode = SKShapeNode(rectOf: CGSize(width: w, height: w),
cornerRadius: w * 0.3)
    self.spinnyNode=spinnyNode
    spinnyNode.position = CGPoint(x: 0.0, y: 0.0)
    spinnyNode.strokeColor = UIColor.red
    spinnyNode.lineWidth = 8.0
    spinnyNode.run(SKAction.sequence([SKAction.wait(forDuration: 0.1),
                                      SKAction.fadeOut(withDuration: 0.1),
                                      SKAction.removeFromParent()]))
    spinnyNode.run(SKAction.repeatForever(SKAction.rotate(byAngle:
6.28, duration: 1)))
     self.run(SKAction.repeatForever(SKAction.sequence([SKAction.w
ait(forDuration: 0.3), SKAction.run({
         let n = spinnyNode.copy() as! SKShapeNode
         self.addChild(n)
          })])))
    }
```

```
    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
     }
}
```

（2）“Game App”创建后会默认包含游戏场景加载代码，笔者增加了中文注释。

```
@IBOutlet var skInterface: WKInterfaceSKScene!
var scene:GameScene?
override func awake(withContext context: Any?) {
    super.awake(withContext: context)
    // Configure interface objects here.
    // Load the SKScene from 'GameScene.sks'
    //从'GameScene.sks'文件加载游戏场景 SKScene
    scene = GameScene(fileNamed: "GameScene")
    // Set the scale mode to scale to fit the window
    //设置视图缩放模式
    scene?.scaleMode = .aspectFill
    // Present the scene
    //显示游戏场景
    self.skInterface.presentScene(scene)
    // Use a value that will maintain a consistent frame rate
    //设置刷新率为 30 帧/秒
    self.skInterface.preferredFramesPerSecond = 30
}
```

（3）触摸屏的操作坐标与 2D 游戏引擎 SpriteKit 的渲染坐标是不同的，所以为了通过触摸屏控制 SpriteKit 的物体位置，需要将屏幕坐标转换为引擎坐标。

```
//屏幕原点为左上角，右下为正；游戏引擎原点为屏幕中心，右上为正。
//42mm 屏幕尺寸大概为(150,168),游戏渲染场景尺寸为(312,390),见 sks 文件
//由屏幕点坐标转换为游戏场景点坐标
```

```
func SCenePoint(sCreenPoint:CGPoint)->CGPoint
{
    let x=(sCreenPoint.x-75)*312/150
    let y=(84-sCreenPoint.y)*390/168
    return CGPoint(x:x , y:y )
}
```

（4）配置单击识别器的操作函数，使光点的位置变换到手指单击的位置。

```
@IBAction func TapGestureRecognizing(_ sender: Any) {
    let tapGesture=sender as? WKTapGestureRecognizer
    let location = tapGesture!.locationInObject()
    let locationText=String(describing:
location.x)+"/"+String(describing: location.y)
    self.setTitle(locationText)
    scene?.spinnyNode?.position=SCenePoint(sCreenPoint: location)
}
```

（5）配置扫动识别器的操作函数，按照扫动的方向改变光点的位置。

```
//扫动手势
@IBAction func SwipeGestureRecognizing(_ sender: Any) {
    let swipeGesture=sender as! WKSwipeGestureRecognizer
    let spinLaction=scene?.spinnyNode?.position
    let direction=swipeGesture.direction
    switch direction{
    case WKSwipeGestureRecognizerDirection.right:
        let newPosition=CGPoint(x: 168, y: (spinLaction?.y)!)
        scene?.spinnyNode?.position=newPosition

    case WKSwipeGestureRecognizerDirection.left:
          let newPosition=CGPoint(x: 0, y: (spinLaction?.y)!)
          scene?.spinnyNode?.position=newPosition
```

```
    case WKSwipeGestureRecognizerDirection.up:
          let newPosition=CGPoint(x: (spinLaction?.x)!, y: 0)
          scene?.spinnyNode?.position=newPosition

    case WKSwipeGestureRecognizerDirection.down:
          let newPosition=CGPoint(x: (spinLaction?.x)!, y: 150)
          scene?.spinnyNode?.position=newPosition
    default:
          ""
    }
}
```

（6）配置拖动识别器的操作函数，拖动操作有不同的状态，如开始、改变、结束、改变，这里我们等拖动完成后将光点移动到结束的位置。

```
@IBAction func PanGestureRecognizing(_ sender: Any) {
    let panGesture=sender as! WKPanGestureRecognizer
    let location = panGesture.locationInObject()
    let locationText=String(describing:
location.x)+"/"+String(describing: location.y)
    self.setTitle(locationText)
    let bounds = panGesture.objectBounds()
    switch panGesture.state {
       case .began:
          break
       case .changed:
          break
       case .cancelled, .ended://拖动结束时才移动
         scene?.spinnyNode?.position=SCenePoint(sCreenPoint: location)
       default:
          debugPrint("Unhandled gesture state: \(panGesture.state)")
       }
}
```

（7）配置长按识别器函数，当长按屏幕时，将文字标签移动到长按的位置。

```
//长按手势
@IBAction func LongPressGestureRecognizing(_ sender: Any) {
    let longPressGesture=sender as! WKLongPressGestureRecognizer
    let location=longPressGesture.locationInObject()
    let locationText=String(describing:
location.x)+"/"+String(describing: location.y)
    self.setTitle(locationText)
    scene?.helloLabel?.position=SCenePoint(sCreenPoint: location)
}
```

5.5 【案例 7】3D 游戏

本案例是苹果官方例子，展示了手表上 3D 游戏引擎控件 WKInterfaceSCNScene 的使用方式。该案例是一个小游戏，玩家可以滑动屏幕旋转一个混乱的模型，当屏幕显示出一个水壶的图形（模型投影）时游戏成功。

1. 场景模型

3D 游戏引擎 SceneKit 会加载一个三维场景文件 sample.scn，放在资源文件夹 art.scnassets 中，该场景文件中含有一个三维模型，从正前面看上去是个茶壶投影，旋转后就会看到散乱的三维模型，如图 5-4 所示。

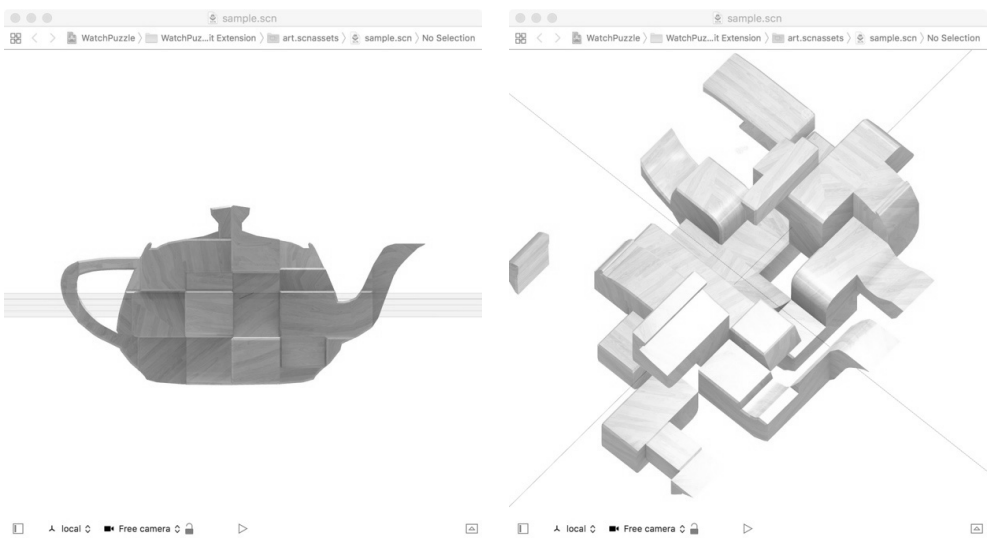


图 5-4　茶壶模型的侧视图和透视图

2. 界面设置

本案例只有一个页面，上面包含一个全屏的 3D 控件“SceneKit Scene”，“SceneKit Scene”右侧属性栏的属性“Scene”设置为 3D 引擎文件“art.scnassets/sample.scn”，然后再绑定两个手势识别器，用于游戏操作，如图 5-5 所示。

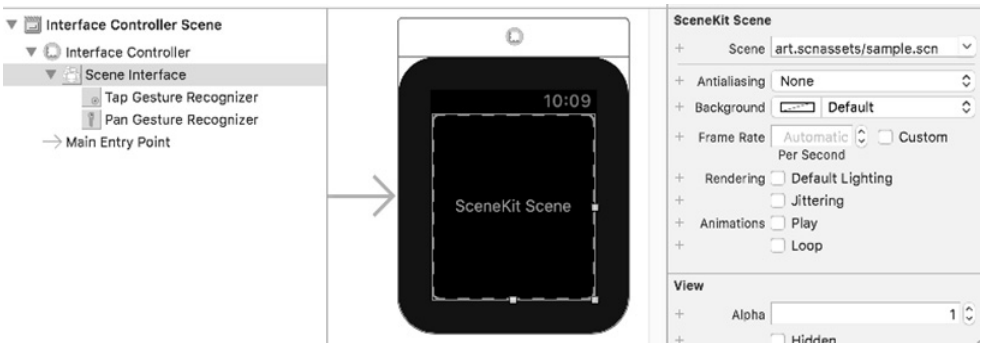


图 5-5　界面设置

3. 开发流程

1）建立 3D 引擎结构体

在 storyboard 里设置“SceneKit Scene”的属性“Scene”为“art.scnassets/sample.scn”，应用运行后会加载 scn 文件并运行，我们只需要在代码中配置和操作 3D 场景就行了。为此，在初始化时建立一个 3D 引擎结构体，引出 3D 场景里所有要操作的物体的句柄：节点、材质、摄像头、文字节点等。

同时，预先定义一些颜色方便供游戏场景里使用。

```
/// A struct containing all the `SCNNode`s used in the game.
    //构建游戏节点 SCNNode 结构体，用于处理游戏
    struct GameNodes {
        let object: SCNNode
        let objectMaterial: SCNMaterial
        let confetti: SCNNode
        let camera: SCNCamera
        let countdownLabel: SKLabelNode
        let congratulationsLabel: SKLabelNode
        /// Queries the root node for the expected nodes.
        //为了方便访问，获取游戏场景里面物体的代表常量
```

```
        init?(sceneRoot: SCNNode) {
            guard let object = sceneRoot.childNode(withName:"teapot",
recursively:true), let objectMaterial = object.geometry?.firstMaterial
else { return nil }
            guard let confetti = sceneRoot.childNode(withName: "particles",
recursively: true) else { return nil }
            guard let camera = sceneRoot.childNode(withName: "camera",
recursively: true)!.camera else { return nil }

            self.object = object
            self.objectMaterial = objectMaterial
            self.confetti = confetti
            self.camera = camera

            countdownLabel = SKLabelNode(fontNamed:"Chalkduster")
            countdownLabel.horizontalAlignmentMode = .center

            congratulationsLabel = SKLabelNode(fontNamed:"Chalkduster");
            congratulationsLabel.fontColor =
InterfaceController.GameColors.defaultFont
            congratulationsLabel.text = "You Win!";
            congratulationsLabel.fontSize = 45;
        }
    }
```

2）初始配置

程序运行开始时进行初始配置，设置游戏初始状态，添加倒计时 2D 文字标签。首先初始化时建议第 1）步的结构体，然后设置转换矩阵、材质和隐藏状态，最后添加 2d 层。

```
  /// Setup overlays and lookup scene objects.
    func setupGame() {
        guard let sceneRoot = sceneInterface.scene?.rootNode, let gameNodes
= GameNodes(sceneRoot: sceneRoot) else { fatalError("Unable to load
game nodes") }
```

```
        self.gameNodes = gameNodes

        gameNodes.object.transform = SCNMatrix4Identity
        gameNodes.objectMaterial.transparency = 0.0
        gameNodes.confetti.isHidden = true

        //设置 2d 层
        let skScene = SKScene(size: CGSize(width: contentFrame.size.width,
height: contentFrame.size.height))
        skScene.scaleMode = SKSceneScaleMode.resizeFill
        skScene.addChild(gameNodes.countdownLabel)
        sceneInterface.overlaySKScene = skScene
    }
```

3）开始游戏

在页面显示时判断游戏状态，如果是非开始状态，则开始运行游戏。游戏开始时将所有物体的状态重置为初始状态，如物体位置朝向、动作、材质显示状态、背景图片、粒子效果、摄像头、重启倒计时等。

```
/// Start the game.
   //开始游戏
   func startGame() {
       guard let gameNodes = gameNodes else { fatalError("Nodes not set") }
       let startSequence = SCNAction.sequence([
           // Wait for 1 second.
           //等 1 秒
           SCNAction.wait(duration: 1.0),
```

```
SCNAction.group([
              // Fade in.
              //淡去
              SCNAction.fadeIn(duration: 0.3),

              // Start the game.
```

```
                //开始游戏
                SCNAction.run({ [weak self] (node: SCNNode) in
                    guard let gameNodes = self?.gameNodes else { return }

                    // Compute a random orientation for the object3D.
                    //计算 3d 物体的朝向
                    let theta = Float(M_PI) * (Float(arc4random()) /
0x100000000)
                    let phi = acosf(2.0 * Float(arc4random()) / 0x
100000000 - 1) / Float(M_PI)
                    var axis = float3()
                    axis.x = cosf(theta) * sinf(phi)
                    axis.y = sinf(theta) * sinf(phi)
                    axis.z = cosf(theta)
                    let angle = 2.0 * Float(M_PI) * (Float(arc4random())
/ 0x100000000)

                    SCNTransaction.begin()
                    SCNTransaction.animationDuration = 0.3
                    SCNTransaction.completionBlock = {
                        self?.gameStarted = true
                    }

                    gameNodes.objectMaterial.transparency = 1.0
                    gameNodes.object.transform = SCNMatrix4MakeRotat
ion(angle, axis.x, axis.y, axis.z)
```

```
                 SCNTransaction.commit()
                }),
            ])
        ])
        gameNodes.object.runAction(startSequence)
```

```
        // Load and set the background image.
        //加载和设置背景图片
        let backgroundImage = UIImage(named:"art.scnassets/backg
round.png")
        sceneInterface.scene?.background.contents = backgroundImage

        // Hide particles, set camera projection to orthographic.
        //隐藏粒子效果，设置摄像头
        particleRemovalTimer?.invalidate()
        gameNodes.congratulationsLabel.removeFromParent()
        gameNodes.confetti.isHidden = true
        gameNodes.camera.usesOrthographicProjection = true

         // Reset the countdown.
        //重启倒计时
        countdown = 30
        gameNodes.countdownLabel.text = "\(countdown)"
        gameNodes.countdownLabel.fontColor =
InterfaceController.GameColors.defaultFont
        gameNodes.countdownLabel.position = CGPoint(x: contentFrame.
size.width / 2, y: contentFrame.size.height - 30)
        textUpdateTimer?.invalidate()
        textUpdateTimer = Timer.scheduledTimer(timeInterval: 1,
                    target: self, selector: #selector(updateText
                    (timer:)), userInfo: nil, repeats: true)
    }
```

4）倒计时报警

倒计 30s，10s 后进行提醒，0s 后报警。

```
/// Update countdown timer.
    //刷新倒计时器
    func updateText(timer: Timer) {
        guard let gameNodes = gameNodes else { fatalError("Nodes not set") }

        gameNodes.countdownLabel.text = "\(countdown)"
```

```
        sceneInterface.isPlaying = true
        sceneInterface.isPlaying = false
        countdown -= 1

        if countdown < 0 {
            gameNodes.countdownLabel.fontColor =
InterfaceController.GameColors.danger
            textUpdateTimer?.invalidate()
            return
        }
        else if countdown < 10 {
            gameNodes.countdownLabel.fontColor =
InterfaceController.GameColors.warning
        }
    }
```

5）旋转物体

通过识别手势识别器进行旋转物体，每次旋转完进行判断是否旋转成功。

```
//手势识别处理
    /**
     通过识别手势的位置进行 3d 物体的旋转
     最后，当方向正确时结束游戏
 */
```

```
 @IBAction func handlePan(panGesture: WKPanGestureRecognizer) {
        guard let gameNodes = gameNodes, gameStarted else { return }
        //手势参数
        let location = panGesture.locationInObject()
        let bounds = panGesture.objectBounds()
        // Compute the projection of the interface point to the virtual
trackball.
        //计算指向虚拟球体界面的投影
        let sphereLocation = sphereProjection(forInterfaceLocation:
location, inBounds: bounds)
```

```
        switch panGesture.state {
            case .began:
                // Record initial states.//记录初始状态
                initialSphereLocation = sphereLocation
                initialObject3DRotation = gameNodes.object.transform
            case .cancelled, .ended, .changed:
   // Compute the rotation and apply to the object.//计算旋转并应用到物体
                let currentRotation =
rotationFromPoint(initialSphereLocation, to: sphereLocation)
                gameNodes.object.transform =
SCNMatrix4Mult(initialObject3DRotation, currentRotation)
            default:
                debugPrint("Unhandled gesture state: \(panGesture.
state)")
        }
        // End the game if the object has the initial orientation.
        //如果物体方向跟最开始的相同，则结束游戏
        if panGesture.state == .ended {
            endGameOnCorrectOrientation()
        }
    }
```

6）便利计算

旋转操作涉及屏幕坐标、引擎 2D 坐标、三维场景坐标，以及旋转、2D 投影等计算，需要有较高的数学基础，一般读者直接套用即可。

```
// MARK: Convenience
    //便利计算工具
    /// Compute the projection of screen points to unit sphere points.
    //计算屏幕到单位球体点的投射
    func sphereProjection(forInterfaceLocation location: CGPoint,
inBounds bounds: CGRect) -> float3 {
        let screenLocation = screenProjection(forInterfaceLocation:
location, inBounds: bounds)
        return sphereProjection(forScreenLocation: screenLocation)
    }
```

```
/// Compute projection from object interface to virtual screen on the
range [-1, 1].
    //计算从物体界面到虚拟屏幕的投射，范围为[-1, 1]
    func screenProjection(forInterfaceLocation location: CGPoint,
inBounds bounds: CGRect) -> CGPoint {
        let w = bounds.size.width
        let h = bounds.size.height
        let aspectRatioCorrection = (h - w) / 2
        var screenCoord = CGPoint(x: location.x / w * 2.0 - 1.0,
                                  y: ((h - location.y) - aspectRatio
Correction) / w * 2.0 - 1.0)
        screenCoord.x = min(1.0, max(-1.0, screenCoord.x))
        screenCoord.y = min(1.0, max(-1.0, screenCoord.y))
        return screenCoord
    }
```

```
/// Compute projection of virtual screen point to unit sphere.
    //计算虚拟屏幕点到单位球体的投射
    func sphereProjection(forScreenLocation location: CGPoint) ->
float3 {
        var sphereCoord = float3()
        let squaredLenght = location.x * location.x + location.y *
location.y

        if squaredLenght <= 1.0 {
            sphereCoord.x = Float(location.x)
            sphereCoord.y = Float(location.y)
            sphereCoord.z = sqrtf(1.0 - Float(squaredLenght))
        } else {
            let n = 1.0 / sqrtf(Float(squaredLenght))
            sphereCoord.x = n * Float(location.x)
            sphereCoord.y = n * Float(location.y)
            sphereCoord.z = 0
        }

        return sphereCoord
    }
```

```
/// Compute the rotation matrix from one point to another on a unit sphere.
    //计算从一点到另一点的单位球体旋转矩阵
    func rotationFromPoint(_ start: float3, to end: float3) ->
SCNMatrix4 {
        let axis = cross(start, end)
        let angle = atan2f(length(axis), dot(start, end))

        return SCNMatrix4MakeRotation(angle, axis.x, axis.y, axis.z)
    }
```

7）判断成功条件

每次旋转后进行判断，如果与原始角度在 10°范围内，则认为是游戏成功，这时结束游戏。

```
/// End the game if the object has its initial orientation with a 10
degree tolerance.
    //如果与原始角度在 10°范围内，则结束游戏
    func endGameOnCorrectOrientation() {
        guard let gameNodes = gameNodes, gameStarted else { return }

        let transform = SCNMatrix4ToMat4(gameNodes.object.transform)
        let unitX: float4 = [1 , 0, 0, 0]
        let unitY: float4 = [0 , 1, 0, 0]
        let tX: float4 = matrix_multiply(unitX, transform)
        let tY: float4 = matrix_multiply(unitY, transform)

        let toleranceDegree : Float = 10.0
        let max_cos_angle = cosf(toleranceDegree * Float(M_PI) / 180)
        let cos_angleX = dot(unitX, tX)
        let cos_angleY = dot(unitY, tY)

        if cos_angleX >= max_cos_angle && cos_angleY >= max_cos_angle {
            endGame()
        }
    }
```

8）结束游戏

符合游戏成功的条件后，进入结束程序，显示白色背景，然后显示祝贺页面。

```
/**
        End the game by showing the congratulation screen after fading
the object to white.
    */
    //结束游戏，在界面淡去变成白色后，显示祝贺界面
```

```
func endGame() {
        guard let gameNodes = gameNodes else { fatalError("Nodes not
set") }

        textUpdateTimer?.invalidate()

        SCNTransaction.begin()
        SCNTransaction.animationDuration = 0.5
        SCNTransaction.completionBlock = { () in
            SCNTransaction.begin()
            SCNTransaction.animationDuration = 0.3
            SCNTransaction.completionBlock = { [weak self] () in
                self?.showCongratulation()
                gameNodes.objectMaterial.emission.contents = UIColor.
black
                self?.gameStarted = false
            }
            SCNTransaction.commit()
        }

        gameNodes.object.transform = SCNMatrix4Identity
        gameNodes.objectMaterial.emission.contents = UIColor.white
        gameNodes.objectMaterial.transparency = 0.0

        SCNTransaction.commit()
    }
```

9）重启游戏

祝贺页面上有重新开始的按钮，单击按钮则重新开始游戏，单击操作通过手势识别器来实现。

```
//按钮事件
    @IBAction func handleTap(sender: AnyObject) {
        if let tapGesture = sender as? WKTapGestureRecognizer {
            if tapGesture.numberOfTapsRequired == 1 && !gameStarted
{
                // Restart the game on single tap only if presenting
congratulation screen.
                //当显示祝贺界面时，单击按钮重启游戏
                startGame()
            }
        }
    }
```

第 6 章

运动传感器和 GPS

6.1 运动传感器

Apple Watch 配备了加速计（accelerometer）和陀螺仪（Gyroscope），我们可以通过 CoreMotion Framework 中的运动管理器 CMMotionManager 来获取相关上述两种传感器的监控数据，传感器可以单独启动，也可以把设备上全部运动传感器一起启动。

1. 加速计

加速计 Accelerometer 负责检测运动的加速度值，分为 X、Y、Z 三个方向。当单独启动加速计时，重力加速度会默认附加在 Z 方向，值为-1.0。启动加速器前，先要设定加速计的监控刷新时间间隔（accelerometerUpdateInterval），每隔一次间隔，加速计就会获取刷新一次数据。获取数据分为两种方式，如果要在获取数据后立即处理，就使用函数回调的方式，即使用 startAccelerometerUpdates(to: withHandler:)，此方法会在后台加速计获取新数据后立即执行 withHandler；如果只是在需要的时候查询数据，就使用 stopAccelerometerUpdates 启动加速器，在需要时读取 CMMotionManager 的属性 accelerometerData 来获取加速数据。检测设备是否支持加速计，可以访问 CMMotionManager 的属性 isAccelerometer Available。

```
//运动管理器
let manager=CMMotionManager()
//检测单项是否可用
if manager.isAccelerometerAvailable//加速计
{
```

```
        aLabel.setTextColor(UIColor.green)
}
//设置刷新间隔
manager.accelerometerUpdateInterval=0.5
//启动加速计，并设置数据执行模块
manager.startAccelerometerUpdates(to: OperationQueue.main) {
```

```
                        (data, error) -> Void in
    print(data!.acceleration.x)
    print(data!.acceleration.y)
    print(data!.acceleration.z)
}
//停止加速计
manager.stopAccelerometerUpdates()
```

2. 陀螺仪

陀螺仪 Gyroscope 负责检测旋转运动的旋转速度，同加速器启动方式相同，先设置间隔 gyroUpdateInterval，再选择调用启动方法 startGyroUpdates(to:withHandler:)或 startGyroUpdates()。

3. 地磁仪

地磁仪 Magnetometer 可以检测磁场强度，但是支持的设备比较少，所以建议使用时要查看 isMagnetometerAvailable 检测本设备是否支持。地磁仪同加速器启动方式相同，先设置间隔 magnetometerUpdateInterval，再选择调用启动方法 startMagnetometerUpdates(to:withHandler:)或 startMagnetometerUpdates()。

4. 设备运动

启动设备运动 Device motion，就表示将设备上所有的运动传感器全部启动，可以获取所有的运动数据，包括重力加速度 gravity、用户加速度 userAcceleration（去掉重力）、姿态 attitude（旋转角度）、旋转速率 rotationRate、磁场 magneticField。

设备运动 Device motion 同加速器启动方式相同，先设置间隔 deviceMotionUpdateInterval，再选择调用启动方法 startDeviceMotionUpdates (using:to:withHandler:)或 startDeviceMotionUpdates(to:withHandler:)或 startDeviceMotionUpdates(using:)或 startDeviceMotionUpdates()，其中 using 参数是姿态矩阵 CMAttitudeReferenceFrame，可以选用。

6.2 传感器记录

watchOS 2.0 开始对传感器记录器 CMSensorRecorder 提供支持，CMSensorRecorder 只提供对加速计数据的记录，使用 recordAccelerometer(forDuration: TimeInterval)设定记录时间段，然后再通过 accelerometerData(from: Date, to: Date)读取记录的数据，详见 6.6 节案例 8。

6.3 运动姿态识别

CoreMotion 框架还包括对设备运动姿态 MotionActivity 的识别，包括静止 stationary、步行 walking、跑步 running、自行车 cycling 的真假状态。MotionActivity 的状态通过 CMMotionActivityManager 的 startActivityUpdates 方法来启动刷新检测，详见 6.6 节案例 8。

6.4 GPS 和定位

Apple Watch Series 2 配备了 GPS，可以直接在手表中调用 CoreLocation Framework 来获取位置信息，与 iOS 中的相同。根据定位服务的使用方式，iOS 10 和 watchOS 3.0 要求在 info.plist 配置相应的权限描述字段，包括以下 3 种：

- Privacy - Location Always Usage Description
- Privacy - Location When In Use Usage Description
- Privacy - Location Usage Description

1）申请权限

使用定位服务，需要向用户申请权限。

```
let locationManager=CLLocationManager()
locationManager.requestWhenInUseAuthorization()
```

2）定位配置

启动定位前，需要对定位精度、过滤距离等参数进行设定。

```
locationManager.desiredAccuracy=kCLLocationAccuracyBestForNavigation
locationManager.distanceFilter=kCLLocationAccuracyKilometer
```

3）获取定位数据

获取定位数据，需要继承代理 CLLocationManagerDelegate 的方法。

```
locationManager(_ manager: CLLocationManager,
 didUpdateLocations locations: [CLLocation])
```

其中，数组 locations 的最后一位是最新的当前位置数据。

CLLocation 包括的数据有二维坐标、海拔、地平面、水平精度、垂直精度、时间戳、速度、方向，具体见表 6-1。

表 6-1　CLLocation 定位数据

属性	类	中文
coordinate	CLLocationCoordinate2D	经纬度
altitude	CLLocationDistance	海拔
floor	CLFloor	地平面
horizontalAccuracy	CLLocationAccuracy	水平精度
verticalAccuracy	CLLocationAccuracy	垂直精度
timestamp	Date	时间戳
speed	CLLocattionSpeed	速度
course	CLLoctionDirection	方向

6.5 地图控件

Apple Watch 提供一个地图控件 WKInterfaceMap，单击控件会弹出系统默认的地图显示页面，该页面显示当前定位的地图视图，但是该视图不能交互。

6.6 【案例 8】运动传感器

本案例演示加速计、陀螺仪和姿态识别器获取运动数据的使用方式（其中有些数据 Apple Watch 是不支持的）。

1. 项目文件目录

本项目主要涉及 3 个文件：传感器主页面 InterfaceController.swif、加速器记录页面 AccelerometerRecorderIC.swift、运动活动状态页面 MotionActivityIC.swift，如图 6-1 所示。

图 6-1　目录结构

2. 界面设置

与文件对应有 3 个页面：传感器数据页面、加速记录页面、运动活动页面，3 个页面并联连接，如图 6-2 所示。

传感器数据页面显示所有运动传感器的数据，包括加速器、陀螺仪、地磁仪。在页面上绑定 4 个 ForceTouch 菜单，用于分别开启不同的传感器。

3. 开发流程

1）功能检测

使用传感器之前要先进行功能检测，查看要使用的传感器是否可用（Available）。

```
override func awake(withContext context: Any?) {
        super.awake(withContext: context)

        // Configure interface objects here.
        //检测单项是否可用
        if manager.isAccelerometerAvailable//加速计
        {
            aLabel.setTextColor(UIColor.green)
```

```
        }
        if manager.isGyroAvailable//陀螺仪、测旋转、姿态
        {
            gLabel.setTextColor(UIColor.green)
        }
        if manager.isMagnetometerAvailable//地磁仪
        {
            mLabel.setTextColor(UIColor.green)
        }
}
```

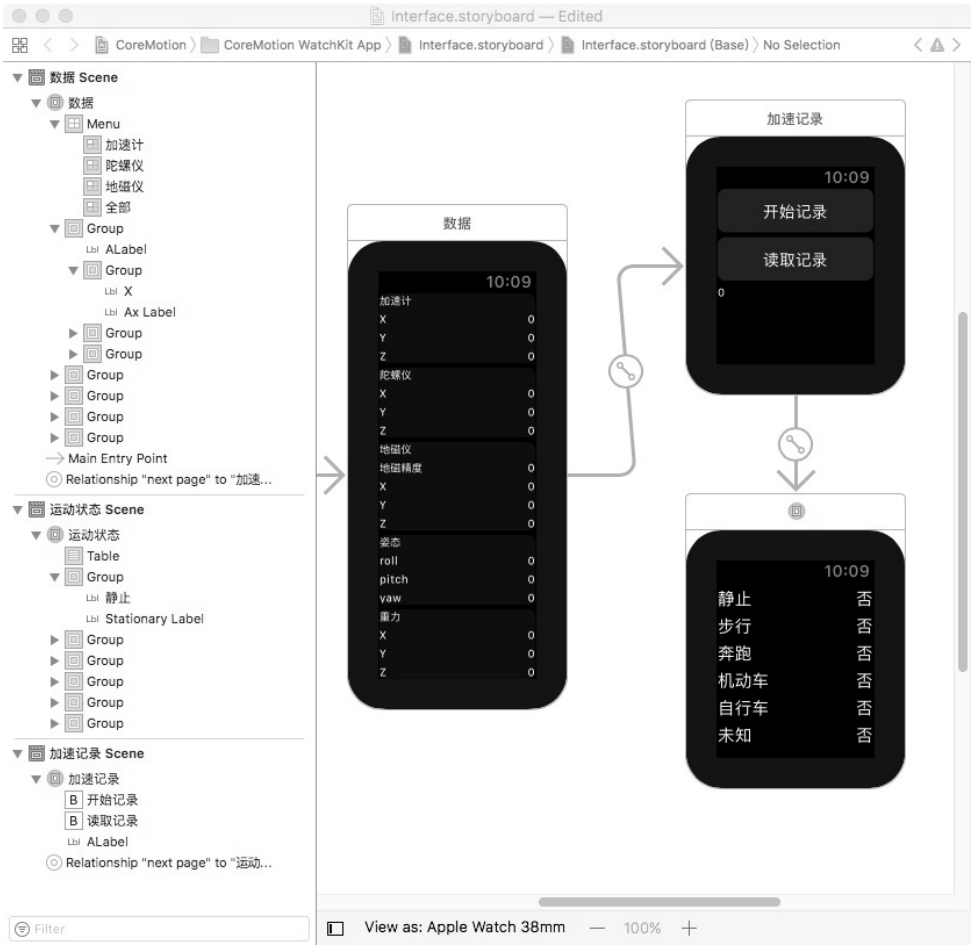


图 6-2　界面设置

2）加速计

加速计是最基本的运动传感器，单独启动加速器时，重力加速度会附加在 *Z* 轴上。

```
@IBAction func UpdateAccelerometer() {

        if aIsUpdating
        {
            aIsUpdating=false
            manager.stopAccelerometerUpdates()
        }else
        {
            aIsUpdating=true

            manager.accelerometerUpdateInterval=0.5
            manager.startAccelerometerUpdates(to: OperationQueue.main) {
                (data, error) -> Void in

                if data==nil
                {
                    return//确保成功获取数据
                }
                self.axLabel.setText(String(data!.acceleration.x))
                self.ayLabel.setText(String(data!.acceleration.y))
                self.azLabel.setText(String(data!.acceleration.z))
            }
        }
}
```

3）陀螺仪

陀螺仪检测旋转运动，结果表示为绕轴旋转的角度，所以可以用来表示设备的姿态朝向。

```
var gIsUpdating=false
    @IBAction func UpdateGyroscope() {
        if gIsUpdating
        {
```

```
            gIsUpdating=false
            manager.stopGyroUpdates()
        }else
        {
            gIsUpdating=true

            manager.gyroUpdateInterval=0.5
            manager.startGyroUpdates(to: OperationQueue.main) {
                (data, error) -> Void in
                if data==nil
                {
                    return//确保成功获取数据
                }
                self.gxLabel.setText(String(data!.rotationRate.x))
                self.gyLabel.setText(String(data!.rotationRate.y))
                self.gzLabel.setText(String(data!.rotationRate.z))
            }
        }
}
```

4）地磁仪

地磁仪检测所处位置的地图磁场分为 3 个方向，通过地磁衰减可以计算出海拔。Apple Watch 对地磁仪的支持不是很好。

```
var mIsUpdating=false
    @IBAction func UpdateMagnetometer() {
        if mIsUpdating
        {
            mIsUpdating=false
            manager.stopMagnetometerUpdates()
        }else
        {
            mIsUpdating=true

            manager.magnetometerUpdateInterval=0.5
```

```
            manager.startMagnetometerUpdates(to: OperationQueue.main) {
                (data, error) -> Void in
                if data==nil
                {
                    return//确保成功获取数据
                }
                self.mxLabel.setText(String(data!.magneticField.x))
                self.myLabel.setText(String(data!.magneticField.y))
                self.mzLabel.setText(String(data!.magneticField.z))
            }
        }
}
```

5）设备运动

通过设备运动 DeviceMotion 方式，可以将上面 3 种传感器全部打开，并读取描述设备运动的数据，如姿态、加速度等，其中加速度不包含重力加速度。

```
var dmIsUpdating=false
    @IBAction func UpdateDeviceMotion() {

        if dmIsUpdating
        {
            dmIsUpdating=false
            //全部停止
            manager.stopAccelerometerUpdates()
            manager.stopMagnetometerUpdates()
            manager.stopGyroUpdates()
            manager.stopDeviceMotionUpdates()
        }else
        {
            dmIsUpdating=true

            manager.deviceMotionUpdateInterval=0.5
            manager.startDeviceMotionUpdates(to: OperationQueue.main) {
                (data, error) -> Void in
                if data==nil
```

```
{
    return//确保成功获取数据
}
//获取所有获得的数据
let attitude=data!.attitude
let rotationRate=data!.rotationRate
let gravity=data!.gravity
let userAcceleration=data!.userAcceleration
let magneticField=data!.magneticField.field
let mAccuracy=data!.magneticField.accuracy

//去掉重力的加速度
self.axLabel.setText(String(userAcceleration.x))
self.ayLabel.setText(String(userAcceleration.y))
self.azLabel.setText(String(userAcceleration.z))

//旋转
self.gxLabel.setText(String(rotationRate.x))
self.gyLabel.setText(String(rotationRate.y))
self.gzLabel.setText(String(rotationRate.z))

//姿态
self.rollLabel.setText(String(attitude.roll))
self.pitchLabel.setText(String(attitude.pitch))
self.yawLabel.setText(String(attitude.yaw))

//地磁
switch mAccuracy//精度
{
case .high:
    self.acLabel.setText("高")
case .medium:
    self.acLabel.setText("中")
case .low:
    self.acLabel.setText("低")
```

```
                case .uncalibrated:
                    self.acLabel.setText("未确定")
                    /*
                default:
                    self.acLabel.setText("未确定")
                    */

                }

                self.mxLabel.setText(String(magneticField.x))
                self.myLabel.setText(String(magneticField.y))
                self.mzLabel.setText(String(magneticField.z))

                //重力
                self.grxLabel.setText(String(gravity.x))
                self.gryLabel.setText(String(gravity.y))
                self.grzLabel.setText(String(gravity.z))

            }
        }
    }
}
```

6）加速计数据记录

CoreMotion 框架还包括传感器记录器，这里演示对加速计数据的记录，记录 10s，然后进行读取记录。

```
//启动记录
   @IBAction func onStart() {

     let operation=BlockOperation()
        { () in
           if CMSensorRecorder.isAccelerometerRecordingAvailable() {
              let recorder = CMSensorRecorder()
              recorder.recordAccelerometer(forDuration: 10) // Record for
10 seconds
```

```
            self.startDate=Date()
          }
      }
      let operationQueue = OperationQueue()
      operationQueue.addOperation(operation)
   }

   //读取记录
   @IBAction func readRecord() {
      if startDate==nil
      {
         return
      }
      if Date().timeIntervalSince(startDate!)<=15
      {
         return
      }
      let recorder = CMSensorRecorder()
         let  list=recorder.accelerometerData(from:  startDate!,  to:
Date())
      var dataString=""
      for listData in list!.enumerated()
      {
         let data=listData as! CMRecordedAccelerometerData
         let xString=String(Float(data.acceleration.x))+"\n"
         let yString=String(Float(data.acceleration.y))+"\n"
         let zString=String(Float(data.acceleration.z))+"\n"
         dataString+=xString+yString+zString+"\n"
      }
      aLabel.setText(dataString)
   }
```

其中 list!.enumerated() 返回的 listData 为 EnumeratedSequence 类型，苹果未提供将其转换为 CMRecordedAccelerometerData 的接口，所以我们自己给它添加 Sequence 接口来完成 as 转换。

```
//给 CMSensorDataList 增加 Sequence 协议，就可以使用 for in 遍历了
extension CMSensorDataList: Sequence {
    public func makeIterator() -> NSFastEnumerationIterator {
        return NSFastEnumerationIterator(self)
    }
}
```

7）运动姿态

CoreMotion 框架还包括对设备运动姿态 MotionActivity 的识别， 这里演示对静止、步行、跑步、自行车真假状态的检测。

```
let activityManager=CMMotionActivityManager()
        activityManager.startActivityUpdates(to: .main){
            activity in
            //初始化
            self.stationaryLabel.setText("否")
            self.walkingLabel.setText("否")
            self.runningLabel.setText("否")
            self.automotiveLabel.setText("否")
            self.cyclingLabel.setText("否")
            self.unknownLabel.setText("否")

            //改变
            if activity?.stationary==true
            {
                self.stationaryLabel.setText("是")
            }
            if activity?.walking==true
            {
                self.walkingLabel.setText("是")
            }
            if activity?.running==true
            {
                self.runningLabel.setText("是")
            }
```

```
            if activity?.automotive==true
            {
                self.automotiveLabel.setText("是")
            }
            if activity?.cycling==true
            {
                self.cyclingLabel.setText("是")
            }
            if activity?.unknown==true
            {
                self.unknownLabel.setText("是")
            }
        }
```

6.7　【案例 9】GPS 定位

本案例演示使用 GPS 定位，获取经纬度及其他相关数据。

（1）项目文件目录，如图 6-3 所示。

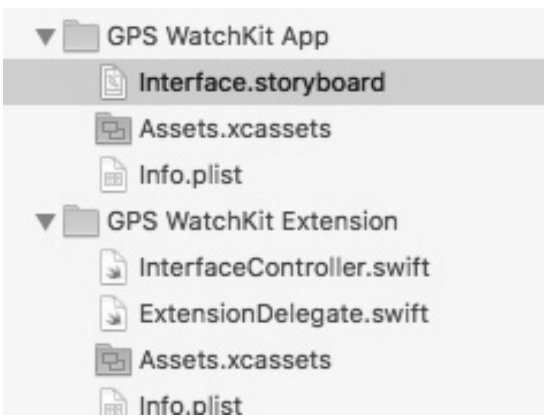


图 6-3　项目文件目录

（2）界面设置，如图 6-4 所示。

图 6-4　界面设置

（3）源代码，如下所示。

```
//  InterfaceController.swift
//  GPS WatchKit Extension
//  Created by 王永超 on 2016/10/15.
//  Copyright © 2016年 王永超. All rights reserved.
import WatchKit
import Foundation
import CoreMotion
class InterfaceController: WKInterfaceController,CLLocationManagerDelegate {
    let locationManager=CLLocationManager()
    @IBOutlet var latitudeLabel: WKInterfaceLabel!
    @IBOutlet var longitudeLabel: WKInterfaceLabel!
    @IBOutlet var autitudeLabel: WKInterfaceLabel!
    @IBOutlet var floorLabel: WKInterfaceLabel!
    @IBOutlet var horizontalAccuracyLabel: WKInterfaceLabel!
```

```
    @IBOutlet var verticalAccuracyLabel: WKInterfaceLabel!
    @IBOutlet var timestampLabel: WKInterfaceLabel!
     override func awake(withContext context: Any?) {
        super.awake(withContext: context)
        // Configure interface objects here.
        locationManager.delegate=self

locationManager.desiredAccuracy=kCLLocationAccuracyBestForNavigation

locationManager.distanceFilter=kCLLocationAccuracyKilometer
        locationManager.requestWhenInUseAuthorization()
    }
      override func willActivate() {
        // This method is called when watch view controller is about to
be visible to user
        super.willActivate()
    }
    override func didDeactivate() {
        // This method is called when watch view controller is no longer
visible
        super.didDeactivate()
    }
    var isStop=true
    @IBAction func startGps()
    {
       if isStop
        {
            locationManager.startUpdatingLocation()
            isStop=false
           }else
        {
            locationManager.stopUpdatingLocation()
            isStop=true
        }
     }
```

```
    func locationManager(_ manager: CLLocationManager,
didUpdateLocations locations: [CLLocation]) {
        let currentLocation=locations.last
        let coordinate2d=currentLocation!.coordinate
        latitudeLabel.setText(String(coordinate2d.latitude))

longitudeLabel.setText(String(coordinate2d.longitude))

autitudeLabel.setText(String(currentLocation!.altitude))

//floorLabel.setText(String(currentLocation!.floor!.level))

horizontalAccuracyLabel.setText(String(currentLocation!.horizontalAcc
uracy))

verticalAccuracyLabel.setText(String(currentLocation!.verticalAcc
uracy))
        timestampLabel.setText(String(describing:
currentLocation!.timestamp))
    }
}
```

第 7 章

与 iPhone 交互

7.1 WatchConnectivity 框架

WatchOS 提供框架 WatchConnectivity framework 进行 Watch 和 iPhone 之间的数据交换，该框架可以后台传输和前台传输。WatchConnectivity framewor 提供了一个 WCSession 对象，通过 WCSession 进行数据传输。

7.2 配置 WCSession

在 Watch Extention 端和 iPhone 端都要先找到默认的 WCSession 对象，指定委托 delegate，并加以激活 activate。

```
if WCSession.isSupported() {
    let session = WCSession.default()
    session.delegate = self
    session.activate()
}
```

7.3 连接状态

7.3.1 判断连接状态

在传输数据时需要判断 Watch 和 iPhone 的连接状态，WCSesson 属性里提供了如下状态：是否配对（isPaired: Bool）、Watch 应用已经安装（isWatchAppInstalled:

Bool)、两者即时消息相通（isReachable: Bool)、WCSession 激活状态(activationState: WCSessionActivationState)。当 Watch 已经配对且 Watch 端应用安装好时，可以进行后台传输数据；当两者消息连接（isReachable: Bool）相通时，可以直接进行前台传输消息。

7.3.2 连接状态回调

指定委托后，就可以在连接状态改变时做出相对反应。当配对或应用安装发生变化时（paired or watchAppInstalled），会调用 sessionReachabilityDidChange()。当即时连接变化时（reachable），会调用 sessionReachabilityDidChange()。我们根据需要，自己在指定的委托中添加上述需要处理的函数。

7.4 数据传输

7.4.1 覆盖式后台传输

后台传输不是数据立刻传输，而是当具备数据传输连接条件以后，Watch 和 iPhone 之间自动同步数据，所以后台传输的数据是异步传输的，具有延后性。

覆盖式后台传输时使用 WCSession 实例的 updateApplicationContext([String : Any])方法。当第一次发送的数据还没有传送出去时，如果此时进行第二次数据传输，会覆盖第一次的数据，而真正传输的是第二次的数据，第一次的数据会丢失。发送后系统会把数据存放在 WCSession 的属性 applicationContext: [String : Any]中。

覆盖式传输是后台异步的，当接收完毕时会调用代理 WCSessionDelegate 的方法 session(_ session: WCSession, didReceiveApplicationContext applicationContext: [String : Any])。系统会把接收的数据存放在 WCSession 的属性 receivedApplication Context: [String : Any]中，我们在需要时读取该属性即可。

7.4.2 队列式后台传输

队列式后台传输，后一次的数据不会覆盖前一次的数据，而是把所有的数据按照次序全部发送出去。队列式后台传输使用 WCSession 的方法 transferUserInfo ([String : Any] = [:])。我们可以通过 WCSession 的属性 outstandingUserInfoTransfers: [WCSessionUserInfoTransfer]对 transferUserInfo 传输进行访问和控制。

队列式传输是后台异步的，当接收完毕时会调用代理 WCSessionDelegate 的方法 session(_ session: WCSession, didReceiveUserInfo userInfo: [String : Any] = [:])。

7.4.3　文件传输

发送文件也是后台异步传输模式，使用 WCSession 的方法 transferFile(URL, metadata: [String : Any]?)。我们可以使用 WCSession 的属性 outstandingFileTransfers: [WCSessionFileTransfer]进行文件传输的访问和管理。

接收文件会后台异步调用代理 WCSessionDelegate 的方法 session(_ session: WCSession, didReceive file: WCSessionFile)。

这里需要注意的是，接收到的文件存放的临时路径中，代理方法 session(_ session: WCSession, didReceive file: WCSessionFile)结束后文件会被系统删除，所以需要立即手动处理（读取或移动）。

另外，手段端文件存储路径应该为组路径，因为组路径可以让 watchkit app 和 extension 同时访问。

7.4.4　消息传输

消息传输时系统会将字典消息尽快以队列式发送传输。

发送消息使用 WCSession 的方法 transferFile(URL, metadata: [String : Any]?)，接受使用代理 WCSessionDelegate 的方法 session(_ session: WCSession, didReceiveMessage message: [String : Any])。

从 Watch 端调用 sendMessage，会唤醒 iPhone 端对应的应用，并使 reachable 为真；但是从 iPhone 端调用 sendMessage，不能唤醒 Watch 端对应的应用，不改变 reachable。所以发送消息时请确保 Watch 和 iPhone 时正常连接，WCSession 的 reachable 状态应为真，且 Watch 打开应用。如果发送消息失败，则调用 errorHandler。

7.4.5　消息数据传输

消息数据传输是以传输消息的方式传输 Data 数据类型。注意，Data 不要太大。发送数据使用 sendMessageData 函数。

7.4.6 功能栏传输

手机端还可以使用专门的函数 transferCurrentComplicationUserInfo 发送用于功能栏的字典到手表端，属于队列式后台传输，手表端通过队列式后台传输的方式来接收字典，使用与之相同的代理方法。

7.5 【案例 10】与 iOS 交互

本案例展示与 iOS 交互的 WatchConnectivity 框架所有种类的传输模式，所有的传输均是手机端发送、手表端接收。

1. 文件目录

本案例的 iOS 端包括主页面文件 ViewController.swift 和一个演示文件发送的数据文件 rtf；watchOS 端只包含一个主页面文件 InterfaceController.swift。整个项目结构如图 7-1 所示。

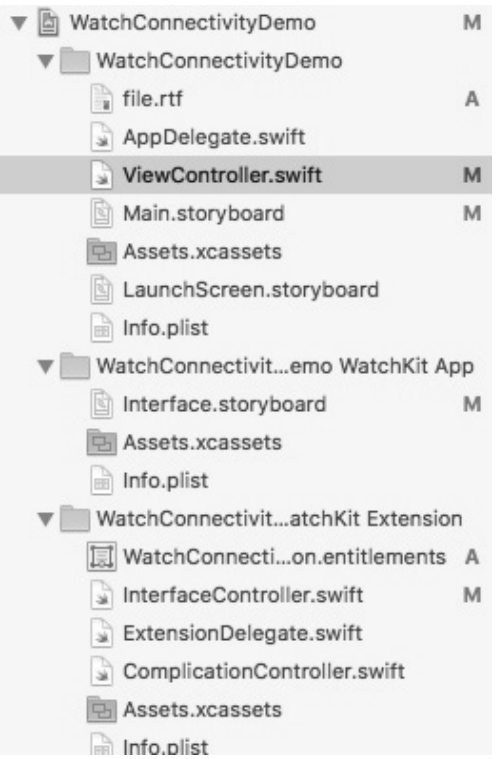


图 7-1 项目目录结构

2. 界面设置

手机界面显示连接状态和各种传输模式的发送按钮，如图 7-2 所示。

图 7-2　手机界面

手表界面包含一个文字标签，显示收到的信息，如图 7-3 所示。

图 7-3　手表界面

3. 开发流程

我们逐个展示 WatchConnectivity 框架的多种传输模式。

1）配置 WCSession

在传输和接受数据之前，要先激活 WCSession 并设置代理。

```
if WCSession.isSupported() {
            let session = WCSession.default()
            session.delegate = self
            session.activate()
 }
```

2）监控连接状态

手机端显示连接状态，使用 WatchConnectivity 的代理方法。

```
func session(_ session: WCSession, activationDidCompleteWith
activationState: WCSessionActivationState, error: Error?)
    {
        switch activationState
        {
        case .activated:
            stateLabel.text="激活"
        case .inactive:
            //从激活到非激活的转变中，可能继续接收数据，但不能发送数据
            stateLabel.text="不活跃"
        case .notActivated:
            stateLabel.text="未激活"
        }
}
 func sessionDidBecomeInactive(_ session: WCSession) {
        stateLabel.text="不活跃"
}
func sessionDidDeactivate(_ session: WCSession) {
        stateLabel.text="未激活"
}
```

3）覆盖式后台传输

手机端通过覆盖式后台发送数据。

```
@IBAction func sendContext(_ sender: Any) {
        let session = WCSession.default()
        let context=["operation":"updateApplicationContext"]//记录操作
        do
        {
            try session.updateApplicationContext(context)//会 throws，要
求使用 do-catch 捕捉
        }catch{

        }
    }
```

手表端通过相应的代理方法接收。

```
//接收到 context
    func session(_ session: WCSession, didReceiveApplicationContext
applicationContext: [String : Any]) {
        let operation=applicationContext["operation"] as! String
        receivedLabel.setText(operation)
    }
```

4）队列式后台传输

手机端通过队列式后台发送数据。

```
//接收到 userInfo//接收到发来的表盘功能栏信息
func session(_ session: WCSession,
             didReceiveUserInfo userInfo: [String : Any] = [:]) {
     let operation=userInfo["operation"] as! String
     receivedLabel.setText(operation)
   }
```

手表端通过相应的代理方法接收。

```
//接收到 context
    func session(_ session: WCSession,
         didReceiveApplicationContext applicationContext: [String :
Any]) {
        let operation=applicationContext["operation"] as! String
```

```
        receivedLabel.setText(operation)
    }
```

5）文件传输

手机端发送文件。

```
 @IBAction func sendFile(_ sender: Any) {
        let session = WCSession.default()
        let metadata=["operation":"transferFile"]
        let mainbundle=Bundle.main
        let fileURL=mainbundle.url(forResource: "file", withExtension:
"rtf")
        session.transferFile(fileURL!, metadata: metadata)//发送
    }
```

手表端通过相应的代理方法接收，并将文件存储在组目录中。

```
//接受到文件
   func session(_ session: WCSession, didReceive file: WCSessionFile) {
      let metadata=file.metadata!
      let operation=metadata["operation"] as! String
      receivedLabel.setText(operation)
      //保存文件,必须存放在组目录中
      let manager = FileManager.default
      let groupcontanerurl=manager.containerURL(
          forSecurityApplicationGroupIdentifier:
          "group.tinghe17.WatchConnectivityDemo")
      let fileName="file.rtf"
      let savefileurl=URL(fileURLWithPath: fileName, isDirectory: false,
relativeTo: groupcontanerurl)
      do {
         //将文件移动到组目录中
         try manager.moveItem(at: file.fileURL, to: savefileurl)
         receivedLabel.setText(operation)
      }catch{

      }
   }
```

6）消息前台传输

手机端发送即时消息。

```
@IBAction func sendMessage(_ sender: Any) {
        let session = WCSession.default()
        let message=["operation":"sendMessage"]//消息
        let replyHander={(reply:[String:Any])->Void in//处理对方返回
的信息

        }
        let errorHander={(error:Error)->Void in//错误处理
            print(error)
        }
        if session.isReachable &&
session.activationState==WCSessionActivationState.activated
        {
            session.sendMessage(message, replyHandler: replyHander,
errorHandler: errorHander)//发送
        }
    }
```

手表端通过相应的代理方法接收。

```
//接收到消息
   func session(_ session: WCSession, didReceiveMessage message: [String :
Any]) {
      let operation=message["operation"] as! String
      receivedLabel.setText(operation)
   }
```

7）消息数据传输

手机端通过消息数据传输的方式发送数据。

```
@IBAction func sendMessageData(_ sender: Any) {
        let session = WCSession.default()

        let data="sendMessageData".data(using: .utf8)
```

```
        let replyHander={(reply:Data)->Void in//处理对方返回的信息

        }
        let errorHander={(error:Error)->Void in//错误处理
            print(error)
        }
        if session.isReachable &&
session.activationState==WCSessionActivationState.activated
        {
            session.sendMessageData(data!, replyHandler: replyHander,
errorHandler: errorHander)//发送
        }

        /*
        let mainbundle=Bundle.main
        let fileURL=mainbundle.url(forResource: "file", withExtension:
"rtf")

        do{
            let data = try Data(contentsOf: fileURL!) //构建 Data


        }catch
        {

        }
        */
 }
```

手表端通过相应的代理方法接收。

```
//接收到数据
     func  session(_  session:  WCSession,  didReceiveMessageData
messageData: Data) {
     let operation=messageData.base64EncodedString()
     receivedLabel.setText(operation)
   }
```

8）功能栏传输

手机端发送功能栏的字典。

```
@IBAction func sendComplication(_ sender: Any) {
        let session = WCSession.default()
        let userInfor=["operation":"transferCurrentComplicationUserInfo"]
        session.transferCurrentComplicationUserInfo(userInfor)
    }
```

手表端通过相应的代理方法接收。

```
//接收到 userInfo//接收到发来的表盘功能栏信息
   func session(_ session: WCSession, didReceiveUserInfo userInfo:
[String : Any] = [:]) {
      let operation=userInfo["operation"] as! String
      receivedLabel.setText(operation)
   }
```

第 8 章

健　康

Apple Watch 提供强大和全面的健康监测功能，如心率、步数、活动能量消耗等，同时 Apple Watch 会将检测到的健康数据先存储在 Watch 端的健康库中；然后会略微延迟地发送到 iPhone 端的健康库中，我们打开 iPhone 上的“健康”应用就能看到检测到的各种数据；最后等过一段时间（大概一天）后，Watch 端会将旧的数据自动删除，以便节省存储空间来存储新数据，通过函数 earliestPermittedSampleDate()可以获取 Watch 端保存的最旧数据的时间。

8.1 健康存储的数据

健康不只是一个简单的应用，而是一个系统级的框架 HealthKit。健康框架 Healthkit 涉及两个应用——“健康”和“健身记录”，其中大部分数据监测数据都可在“健康”中访问，而“健身记录”可以访问有关“体能训练”的数据。为了更直观地认识，我们可以打开 iPhone 上的“健康”和“健身记录”应用。

健康框架 HealthKit 具有一个健康库 HealthStore，此 HealthStore 负责健康数据的存储、管理和访问，同时也负责健康传感器的管理工作。

HealthStore 存储的数据包括人体特征数据、样本数据和病历等。

8.1.1 人体特征数据

人体特征数据（Characteristic data）是指人体不随时间发生变化的数据，如生日 / 年龄、血型、性别等。特征数据可以通过 HealthStore 直接获得，不必经过复杂查询操作。

8.1.2 样本数据

样本数据（Sample data）是指随时间变化的数据，如心率、行走步数、能量消耗等，类名为 HKSample。 HKSample 具有对应的开始时间、结束时间和数据类型。

HKSample 可分为四个子类：类别样本 HKCategorySample、数量样本 HKQuantitySample、关联样本 HKCorrelation 和体能训练 HKWorkout。

类别样本 HKCategorySample 是以状态变化为标志的数据，如睡眠的睡着与醒着的两种状态。

数量样本 HKQuantitySample 是数值表示的数据，如心率为 65bpm。

关联样本 HKCorrelation 是将不同的数据关联起来放在一起，形成一个记录的数据，如一个关联样本，这个数据包括心率和能量消耗两个数据。

体能训练 HKWorkout 表示一段时间的运动记录，一般包括运动类型、运动位置、起始时间、距离、能量消耗、步数、平均心率及 metaData 数据，其中运动类型、运动位置、起始时间、距离、能量消耗是 HKWorkout 必须要包含的基本数据，步数、平均心率、metaData 等其他数据是额外附加数据。手机端“健身记录”里有关户外运动显示地图和天气，可能是以 metaData 形式存储的。HKWorkout 通过 Watch 端的体能训练作业 HKWorkoutSession 来监测，HKWorkoutSession 可在 Watch 端创建和激活，激活后 Watch 的健康传感器就会开始工作（耗电较多），全面检测步数、能量消耗、心率、步数等数据，并自动将检测到的数据以 HKQuantitySample 的形式存储起来并发送到手机端。我们需要手动将体能训练 HKWorkout 及其相关的数据保存到健康库 HealthStore 中，存储的体能训练 HKWorkout 可以在手机“健身记录”里查看，存储的体能训练 HKWorkout 能增加运动环数据。

8.1.3 样本数据类型

为了区别各种数据的不同类型，健康库 HealthStore 通过健康数据类型 HKObjectType 来表示数据的类型，而 HKObjectType 又是通过相应的类型标志 Identifier（枚举类型）来确定的，如表 8-1 所示。

表 8-1　样本数据类型

Identifier 枚举	Identifier 值	中文名称
HKCharacteristicTypeIdentifier	biologicalSex	性别
	bloodType	血型
	dateOfBirth	出生日期
	fitzpatrickSkinType	皮肤类型
	wheelchairUse	轮椅使用
HKCategoryTypeIdentifier	appleStandHour	站立时间
	cervicalMucusQuality	宫颈黏液质量
	intermenstrualBleeding	经期出血
	menstrualFlow	月经周期
	mindfulSession	焦虑期
	ovulationTestResult	排卵测试结果
	sexualActivity	性活动
	sleepAnalysis	睡眠分析
HKCorrelationTypeIdentifier	bloodPressure	血压
	food	食物
HKQuantityTypeIdentifier	activeEnergyBurned	活动能量燃烧
	appleExerciseTime	活动时间
	basalBodyTemperature	基础体温
	basalEnergyBurned	基础能量燃烧
	bloodAlcoholContent	血液酒精含量
	bloodGlucose	血糖
	bloodPressureDiastolic	血压舒张
	bloodPressureSystolic	血压收缩压
	bodyFatPercentage	体脂百分比
	bodyMass	体重
	bodyMassIndex	体质指数
	bodyTemperature	体温
	dietaryBiotin	饮食生物素
	dietaryCaffeine	饮食咖啡因
	dietaryCalcium	饮食钙
	dietaryCarbohydrates	饮食碳水化合物
	dietaryChloride	饮食氯
	dietaryCholesterol	饮食胆固醇
	dietaryChromium	饮食铬

（续表）

Identifier 枚举	Identifier 值	中文名称
HKQuantityTypeIdentifier	dietaryCopper	饮食铜
	dietaryEnergyConsumed	饮食能量消耗
	dietaryFatMonounsaturated	饮食单不饱和脂肪
	dietaryFatPolyunsaturated	饮食多不饱和脂肪
	dietaryFatSaturated	饮食饱和脂肪
	dietaryFatTotal	饮食总脂肪
	dietaryFiber	饮食纤维
	dietaryFolate	饮食叶酸
	dietaryIodine	饮食碘
	dietaryIron	饮食铁
	dietaryMagnesium	饮食镁
	dietaryManganese	饮食锰
	dietaryMolybdenum	饮食钼
	dietaryNiacin	饮食烟酸
	dietaryPantothenicAcid	饮食遍多酸
	dietaryPhosphorus	饮食磷
	dietaryPotassium	饮食钾
	dietaryProtein	饮食蛋白质
	dietaryRiboflavin	饮食核黄素
	dietarySelenium	饮食硒
	dietarySodium	饮食钠
	dietarySugar	饮食糖
	dietaryThiamin	饮食维生素
	dietaryVitaminA	饮食维生素 A
	dietaryVitaminB12	饮食维生素 B_{12}
	dietaryVitaminB6	饮食维生素 B_6
	dietaryVitaminC	饮食维生素 C
	dietaryVitaminD	饮食维生素 D
	dietaryVitaminE	饮食维生素 E
	dietaryVitaminK	饮食维生素 K
	dietaryWater	饮食水
	dietaryZinc	饮食锌
	distanceCycling	单车距离
	distanceWalkingRunning	走跑距离
	distanceWheelchair	轮椅距离
	electrodermalActivity	皮肤电活动
	flightsClimbed	航班上

（续表）

Identifier 枚举	Identifier 值	中文名称
HKQuantityTypeIdentifier	forcedExpiratoryVolume1	用力呼气量
	forcedVitalCapacity	用力肺活量
	heartRate	心率
	height	高度
	inhalerUsage	吸入器使用
	leanBodyMass	瘦体重
	nikeFuel	耐克燃料
	numberOfTimesFallen	下降次数
	oxygenSaturation	氧饱和度
	peakExpiratoryFlowRate	呼吸率峰值
	peripheralPerfusionIndex	末梢灌注指数
	pushCount	推数
	respiratoryRate	呼吸率
	stepCount	步数
	uvExposure	紫外曝光
HKDocumentTypeIdentifier	CDA	病历

8.1.4 数据单位

HKQuantitySample 是数据量，涉及所对应的数量单位 HKUnit，如米、次/分钟等，其中米可由函数 HKUnit.meter()得到，次/分钟则通过字符串构造 HKUnit(from: " count/min ")。

8.1.5 病历

健康库 HealthStore 可以通过第三方应用添加病历文件 CDA，用户可以通过“健康”应用查看和管理病历文件。因为病历主要通过手机操作，所以本书不涉及病历 CDA 的开发内容。

8.2 监测数据

8.2.1 加载健康框架

Apple 产品使用健康功能是通过健康框架（HealthKit Framework）实现的，所

以我们先要将 HealthKit Framework 添加到手机项目和手表 Extension 项目中，在 xcode 项目中的“Capabilities”开启“HealthKit”即可。

8.2.2　申请权限

Apple Watch 最主要的功能就是监测健康数据，由于健康涉及个人隐私，所以需要申请相应的权限，申请权限时要告诉系统要求读写的数据类别，申请权限的代码必须写在 iPhone 端，同时必须在 info.plist 中添加键值 NSHealthUpdateUsage Description（Privacy - Health Update Usage Description）和 NSHealthShareUsage Description（Privacy - Health Share Usage Description）。

```
let set:Set<HKSampleType> = [
    HKSampleType.quantityType(
      forIdentifier: HKQuantityTypeIdentifier.distanceWalkingRunning)!,
    HKSampleType.quantityType(
      forIdentifier: HKQuantityTypeIdentifier.activeEnergyBurned)!,
    HKSampleType.quantityType(
      forIdentifier: HKQuantityTypeIdentifier.stepCount)!,
    HKSampleType.quantityType(
      forIdentifier: HKQuantityTypeIdentifier.heartRate)!,
    HKSampleType.workoutType()
      ]
let healthStore = HKHealthStore()
       healthStore.requestAuthorization(toShare: set, read:set )
       {(bool,error) in


       }
```

8.2.3　后台模式

苹果手表应用在使用时很可能会黑屏或者切换到时间表盘，此时应用一般会被挂起暂停运行，所以为了能够持续不断地检测，应用需要开启后台运行模式，在 xcode 的 Extension 项目的“Capabilities”打开“Background Modes”，勾选其中的“Workout Processing”。

8.2.4 监测体能训练

成功申请权限后，Watch 端 HealthStore 就可以启动体能训练作业 WorkoutSession，全面激活健康传感器，之后也可以进行暂停/继续和停止操作。WorkoutSession 开始时需要设定运动的类型 HKWorkoutActivityType 和位置（室内/室外），进行不同的运动，检测到的数据是不同的。WorkoutSession 的运行状态通过代理 HKWorkoutSessionDelegate 来调用状态改变方法 workoutSession(_:didChange To:from:date:)。

未开始 notStarted、运行 running、暂停 paused、结束 ended 是传感器状态 HKWorkoutSessionState 的 4 种变化，我们要根据不同状态进行不同的后续操作。

WorkoutSession 在工作时（运行 running），健康传感器获取新检测数据会自动存储在当前设备的健康库中，然后通过锚点查询（HKAnchoredObjectQuery）方式进行函数回调处理返回的新数据。查询数据时要告知系统数据的类型、过滤条件 predicate（如时间范围、存储设备、附加信息）、查询锚点 HKQueryAnchor、数量限制等信息。第一次运行时，查询锚点 HKQueryAnchor 设置为 nil，查询会返回健康库中符合条件的所有数据（确保为最新数据，将时间设置为启动开始）；在后续查询中，会使用前一次查询返回的查询锚点 HKQueryAnchor，所以后续查询只返回前一次查询以后新添加的数据。

```
let workoutConfiguration = HKWorkoutConfiguration()
      workoutConfiguration.activityType = .walking
      workoutConfiguration.locationType = .outdoor
           do {
         workoutSession  =  try  HKWorkoutSession(configuration:
workoutConfiguration)
         workoutSession?.delegate = self
         workoutStartDate = Date()
         healthStore.start(workoutSession!)
      } catch {
         // ...error
      }
func workoutSession(_ workoutSession: HKWorkoutSession,
                    didChangeTo toState: HKWorkoutSessionState,
```

```
                    from fromState: HKWorkoutSessionState,
                    date: Date) {
        switch toState {
        case .running:
            //原来没有开始，现在开始是第一次
if fromState == .notStarted {
    // 开始查询数据
                startAccumulatingData()
            } else {
                //继续
            }
            break
        case .paused:
              //暂停
            break
        case .ended:
            //结束查询，然后保存
              stopAccumulatingData()
            saveWorkout()
            break
        default:
            break
        }
    }
func startAccumulatingData() {
        //最少查询两种类型的数据
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.distanceWalkingRunning)
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.activeEnergyBurned)
    }
    func startQuery(quantityTypeIdentifier: HKQuantityTypeIdentifier) {
        //配置查询
        let datePredicate = HKQuery.predicateForSamples(withStart:
workoutStartDate, end: nil, options: .strictStartDate)
```

```
        let devicePredicate = HKQuery.predicateForObjects(from:
[HKDevice.local()])
        let queryPredicate =
NSCompoundPredicate(andPredicateWithSubpredicates:[datePredicate,
devicePredicate])

        //获得新数据时调用的函数
        let updateHandler: ((HKAnchoredObjectQuery, [HKSample]?,
[HKDeletedObject]?, HKQueryAnchor?, Error?) -> Void) = { query, samples,
deletedObjects, queryAnchor, error in
            //处理得到的数据
            self.process(samples:  samples,  quantityTypeIdentifier:
quantityTypeIdentifier)
        }
        //构建锚点查询，要求获取检测到的最新数据
        let query = HKAnchoredObjectQuery(type:
HKObjectType.quantityType(forIdentifier: quantityTypeIdentifier)!,
                                          predicate: queryPredicate,
                                          anchor: nil,
                                          limit: HKObjectQueryNoLimit,
                                          resultsHandler: updateHandler)
        //开始执行查询
        healthStore.execute(query)
        //将查询放到全局序列中，已被后续操作
        activeDataQueries.append(query)
    }
    func  process(samples:  [HKSample]?,  quantityTypeIdentifier:
HKQuantityTypeIdentifier) {
        if let quantitySamples = samples as? [HKQuantitySample] {
            for sample in quantitySamples {
                switch quantityTypeIdentifier
                {
                //距离
                case HKQuantityTypeIdentifier.distanceWalkingRunning:
                    let newMetersValue = sample.quantity.doubleValue
```

```
(for: HKUnit.meter())//转成值
                let totalDistanceValue=totalDistance.doubleValue(for:
HKUnit.meter())+newMetersValue
                totalDistance=HKQuantity(unit: HKUnit.meter(),
doubleValue:totalDistanceValue)//转换成 HKQuantity
                //显示,调用前端线程
                DispatchQueue.main.sync {

distanceLabel.setText(String(Int(newMetersValue))+" / "+String(Int
(totalDistanceValue)))
                }
            //能量
            case HKQuantityTypeIdentifier.activeEnergyBurned:
                let newKCalValue = sample.quantity.doubleValue(for:
HKUnit.kilocalorie())
                let
totalEnergyBurnedValue=totalEnergyBurned.doubleValue(for:
HKUnit.kilocalorie())+newKCalValue
                totalEnergyBurned=HKQuantity(unit:HKUnit.kilocalorie(),
doubleValue: totalEnergyBurnedValue)
                //显示
                DispatchQueue.main.sync {

energyLabel.setText(String(Int(newKCalValue))+" / "+String(Int
(totalEnergyBurnedValue)))
                }
            default:break
            }
        }
    }
}
```

8.2.5 活动类型

watchOS 对大量活动类型 HKWorkoutActivityType 提供了支持，详细如表 8-2 所示。

表 8-2　HKWorkoutActivityType 支持类型

HKWorkoutActivityType	中文名称	HKWorkoutActivityType	中文名称
americanFootball	美式足球	archery	射箭
australianFootball	澳式足球	badminton	羽毛球
baseball	棒球	basketball	篮球
bowling	保龄球	boxing	拳击
climbing	攀登	cricket	板球
crossTraining	交叉培训	curling	冰壶
cycling	单车	dance	舞蹈
elliptical	椭圆机	equestrianSports	马术
fencing	击剑	fishing	钓鱼
functionalStrengthTraining	功能性力量训练	golf	高尔夫
gymnastics	体操	handball	手球
hiking	徒步	hockey	曲棍球
hunting	狩猎	lacrosse	长曲棍球
martialArts	武术	mindAndBody	身心
mixedMetabolicCardioTraining	混合代谢心脏训练	paddleSports	皮划艇
play	玩	preparationAndRecovery	准备恢复
racquetball	壁球	rowing	划船
rugby	橄榄球	running	跑步
sailing	帆船	skatingSports	滑冰
snowSports	滑雪	soccer	足球
softball	软球	squash	墙网球
stairClimbing	爬楼梯	surfingSports	冲浪
swimming	游泳	tableTennis	乒乓器
tennis	网球	trackAndField	田径
traditionalStrengthTraining	传统力量训练	volleyball	排球
walking	步行	waterFitness	水中健身
waterPolo	水球	waterSports	水上运动
wrestling	摔跤	yoga	瑜伽
other	其他	barre	芭蕾把杆健身
coreTraining	核心训练	crossCountrySkiing	越野滑雪
downhillSkiing	高山滑雪	flexibility	柔韧性
highIntensityIntervalTraining	高强度间歇训练	jumpRope	跳绳
kickboxing	自由搏击	pilates	普拉提
snowboarding	滑雪板	stairs	楼梯
stepTraining	步训练	wheelchairWalkPace	轮椅行走步伐
wheelchairRunPace	轮椅跑步步伐		

8.2.6　存储到健康库

我们需要将检测到的数据绑定到体能训练 HKWorkout，然后将 HKWorkout 保存到健康库 HealthStore 中，存储的体能训练 HKWorkout 可以在手机“健身记录”里查看。创建体能训练默认需要总距离和总能量，至于平均心率等其他数据可以在体能训练保存后添加到体能训练上，详见 8.3 节案例 11。

```
func stopAccumulatingData() {
       for query in activeDataQueries {
          //停止查询
          healthStore.stop(query)
       }
       activeDataQueries.removeAll()
    }
func saveWorkout() {
       // 创建和保存 workout
       let configuration = workoutSession!.workoutConfiguration
       let isIndoor = (configuration.locationType == .outdoor) as NSNumber

      let workout = HKWorkout(activityType: configuration.)activityType,
                              start: workoutStartDate!,
                              end: workoutEndDate!,
                              workoutEvents: workoutEvents,
                              totalEnergyBurned: totalEnergyBurned,
                              totalDistance: totalDistance,
                              metadata:
[HKMetadataKeyIndoorWorkout:isIndoor]);//附加信息，元数据

       healthStore.save(workout) { success, _ in
          if success {
             //这里添加其他数据


    }
          }
```

```
        }
}
```

8.3 【案例 11】健身监测和体能训练

本案例在手表上使用健康框架检测常规的健身数据能量、距离、步数和心率，并将监测到的数据以体能训练的形式保存到健康库中，在手机应用“健身记录”中的“体能训练”中可以查看保存的结果。

1. 项目文件和文件

本项目主要涉及手机端主页面文件 ViewController.swift，手表端主页面文件 InterfaceController.swift。手机页面没有内容，手表页面包含 4 种数据显示和控制按钮，如图 8-1 所示。

图 8-1　手表页面设置

2. 开发流程

1）申请健康权限

在手机端 ViewController 的初始化 viewDidLoad()中申请健康权限，包括能量、距离、步数和心率 4 种数据的读取和写入。

```
//申请权限
        let set:Set<HKSampleType> =
[HKSampleType.quantityType(forIdentifier:
HKQuantityTypeIdentifier.distanceWalkingRunning)!,
                                    HKSampleType.quantityType(forIde
ntifier: HKQuantityTypeIdentifier.activeEnergyBurned)!,
                                    HKSampleType.quantityType(forIde
ntifier: HKQuantityTypeIdentifier.stepCount)!,
                                    HKSampleType.quantityType(forIde
ntifier: HKQuantityTypeIdentifier.heartRate)!,
                                    HKSampleType.workoutType()
        ]
        let healthStore = HKHealthStore()
        healthStore.requestAuthorization(toShare: set, read:set )
        {(bool,error) in


        }
```

2）声明成员变量

在执行函数之前，先声明健康监测使用的成员变量，主要涉及 workout 和盛放数据的变量，以方便在整个流程中使用。

```
// 健康相关成员变量
    let healthStore = HKHealthStore()

    //workout
    var workoutSession : HKWorkoutSession?
    var workoutStartDate : Date?
    var workoutEndDate : Date?
    var workoutEvents = [HKWorkoutEvent]()
    var metadata = [String: Any]()

    //numbers
    var activeDataQueries = [HKQuery]()
    var totalEnergyBurned = HKQuantity(unit: HKUnit.kilocalorie(),
doubleValue: 0)
```

```
    var totalDistance = HKQuantity(unit: HKUnit.meter(), doubleValue: 0)
    var totalCount = HKQuantity(unit: HKUnit.count(), doubleValue: 0)
    var averageHeartRate = HKQuantity(unit: HKUnit(from: "count/min"),
doubleValue: 0)
    var count=0.0//检测次数
```

3）开启健康传感器

开启健康传感器前，要先设置活动类型和场景，记录开启时间，并配置代理。

```
    @IBAction func onStart() {
        //配置 workout
        let workoutConfiguration = HKWorkoutConfiguration()
        workoutConfiguration.activityType = .walking//类型
        workoutConfiguration.locationType = .outdoor//环境
        //捕捉模式
        do {
            //创建 HKWorkoutSession
              workoutSession = try HKWorkoutSession(configuration:
workoutConfiguration)
            workoutSession?.delegate = self//设置代理

            workoutStartDate = Date()
            //启动，传感器开始工作
            healthStore.start(workoutSession!)
        } catch {
            // ...error
        }
      }
```

4）暂停健康传感器

开启健康传感器后可以暂停健康传感器，暂停后还可以继续。

```
   @IBAction func onPauseResume() {
        //根据不同的状态采取反向操作
        if let session = workoutSession {
            switch session.state {
            case .running:
```

```
                healthStore.pause(_: session)
            case .paused:
                healthStore.resumeWorkoutSession(_: session)
            default:
                break
            }
        }
    }
```

5）停止健康传感器

结束监测时要停止 WorkoutSession。

```
@IBAction func onStop() {
        workoutEndDate = Date()

        // 结束 Workout Session
        healthStore.end(workoutSession!)
    }
```

6）传感器状态变化

未开始 notStarted、运行 running、暂停 paused、结束 ended 是传感器状态 HKWorkoutSessionState 的 4 种变化，我们要根据不同状态进行不同的后续操作，如从未开始变为运行是启动状态，此刻开始获取测量的数据；当变为停止状态时，即刻停止查询数据，然后将收集到的数据保存起来。

```
func workoutSession(_ workoutSession: HKWorkoutSession,
                      didChangeTo toState: HKWorkoutSessionState,
                      from fromState: HKWorkoutSessionState,
                      date: Date) {
        switch toState {
        case .running:
            //原来没有开始，现在开始是第一次
            if fromState == .notStarted {
                //开始积攒数据
                startAccumulatingData()
            } else {
                //继续积攒数据
```

```
                resumeAccumulatingData()
            }
            break
        case .paused:
            //暂停积攒数据
            pauseAccumulatingData()
            break
        case .ended:
            //停止积攒数据
            stopAccumulatingData()
            //保存 workout
            saveWorkout()
            break
        default:
            break
        }

    }
func stopAccumulatingData() {
        for query in activeDataQueries {
            //停止查询
            healthStore.stop(query)
        }

        activeDataQueries.removeAll()
    }

    func pauseAccumulatingData() {
        DispatchQueue.main.sync {
            isPaused = true
        }
    }

    func resumeAccumulatingData() {
        DispatchQueue.main.sync {
```

```
            isPaused = false
        }
    }
```

7）开始积攒数据

积攒数据就是持续地将新检测的数据记录起来的过程。要获取新检测的数据，可以采用锚点查询（HKAnchoredObjectQuery）方式，在健康传感器获取新检测数据时会自动存储在当前设备的健康库中，然后锚点查询（HKAnchoredObjectQuery）方式会进行处理函数回调，并返回新数据。查询数据时要告知系统数据的类型、过滤条件（如时间为从 workout 启动时算起、本地设备）、锚点等信息，同时要配置回调函数来处理查询到的新数据。这里我们查询 4 种数据，分别是距离、能量消耗、步数、心率。

```
func startAccumulatingData() {
        //查询 4 种类型的数据
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.distanceWalkingRunning)
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.activeEnergyBurned)
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.stepCount)
        startQuery(quantityTypeIdentifier:
HKQuantityTypeIdentifier.heartRate)
    }

    func startQuery(quantityTypeIdentifier: HKQuantityTypeIdentifier) {
        //配置查询
        let datePredicate = HKQuery.predicateForSamples(withStart:
workoutStartDate, end: nil, options: .strictStartDate)
        let devicePredicate = HKQuery.predicateForObjects(from:
[HKDevice.local()])
        let queryPredicate =
NSCompoundPredicate(andPredicateWithSubpredicates:[datePredicate,
devicePredicate])

        //获得新数据时调用的函数
```

```
        let updateHandler: ((HKAnchoredObjectQuery, [HKSample]?,
[HKDeletedObject]?, HKQueryAnchor?, Error?) -> Void) = { query, samples,
deletedObjects, queryAnchor, error in
            //处理得到的数据
            self.process(samples: samples, quantityTypeIdentifier:
quantityTypeIdentifier)
        }
        //构建锚点查询，要求获取检测到的最新数据
        let query = HKAnchoredObjectQuery(type:
HKObjectType.quantityType(forIdentifier: quantityTypeIdentifier)!,
                                          predicate: queryPredicate,
                                          anchor: nil,
                                          limit: HKObjectQueryNoLimit,
                                          resultsHandler: updateHandler)
        //开始执行查询
        healthStore.execute(query)
        //将查询放到全局序列中，已被后续操作
        activeDataQueries.append(query)
    }
```

8）加工处理数据

锚点查询可以得到最新的样本数据，这里演示将新数据积累起来进行加工处理，得到总能量、总距离、总步数和平均心率。监测时还需更新界面显示数据。

```
func process(samples: [HKSample]?, quantityTypeIdentifier:
HKQuantityTypeIdentifier) {
        if let quantitySamples = samples as? [HKQuantitySample] {
            for sample in quantitySamples {
                switch quantityTypeIdentifier
                {
                //距离
                case HKQuantityTypeIdentifier.distanceWalkingRunning:
                    let newMetersValue = sample.quantity.doubleValue(for:
HKUnit.meter())//转成值
                    let totalDistanceValue=totalDistance.doubleValue
(for:HKUnit.meter())+newMetersValue
```

```
                    totalDistance=HKQuantity(unit: HKUnit.meter(),
doubleValue: totalDistanceValue)//转换成 HKQuantity
                    //显示,调用前端线程
                    DispatchQueue.main.sync {
                        distanceLabel.setText(String(Int(newMetersVa
lue))+" / "+String(Int(totalDistanceValue)))
                    }
                //能量
                case HKQuantityTypeIdentifier.activeEnergyBurned:
                    let newKCalValue = sample.quantity.doubleValue(for:
HKUnit.kilocalorie())
                    let
totalEnergyBurnedValue=totalEnergyBurned.doubleValue(for:
HKUnit.kilocalorie())+newKCalValue
                    totalEnergyBurned=HKQuantity(unit: HKUnit.
kilocalorie(), doubleValue: totalEnergyBurnedValue)
                    //显示
                    DispatchQueue.main.sync {
                       energyLabel.setText(String(Int(newKCalValue))
+" / "+String(Int(totalEnergyBurnedValue)))
                    }

                //步数
                case HKQuantityTypeIdentifier.stepCount:
                    let newStepCountValue = sample.quantity.doubleValue
(for: HKUnit.count())
                     let totalCountValue=totalCount.doubleValue(for:
HKUnit.count())+newStepCountValue
                    totalCount=HKQuantity(unit: HKUnit.count(),
doubleValue: totalCountValue)
                    //显示
                    DispatchQueue.main.sync {
                        countLabel.setText(String(Int(newStepCountVa
lue))+" / "+String(Int(totalCountValue)))
                    }
```

```
                //心率
                case HKQuantityTypeIdentifier.heartRate:

                    let newHeartRateValue = sample.quantity.doubleValue
(for: HKUnit(from: "count/min"))
                    var
averageHeartRateValue=averageHeartRate.doubleValue(for: HKUnit(from:
"count/min"))
                    averageHeartRateValue=(averageHeartRateValue*count+
newHeartRateValue)/(count+1)
                    count+=1
                    averageHeartRate=HKQuantity(unit: HKUnit(from: "
count/min"), doubleValue: averageHeartRateValue)
                    //显示
                    DispatchQueue.main.sync {
                        heartRateLabel.setText(String(Int(newHeartRa
teValue))+"/"+String(Int(averageHeartRateValue)))
                    }

                default:break
                }

            }

            // strongSelf.updateLabels()

        }
    }
```

9）保存为体能训练

停止检测后，我们将有关监测的信息（时间段、总能量、总距离）绑定到一个体能训练 workout 上，并将该体能训练 workout 保存到健康库中，最后还可以把总步数和平均心率添加到该体能训练中。至于查询的 4 种数据样本（距离、能量消耗、步数、心率）已经由系统自动保存在健康库中，锚点查询得到的是已经保存后的数据，我们需要样本数据时直接查询健康库即可。

```
func saveWorkout() {
        // 创建和保存 workout
        let configuration = workoutSession!.workoutConfiguration
        let isIndoor = (configuration.locationType == .outdoor) as NSNumber


       let workout = HKWorkout(activityType: configuration.activityType,
                                start: workoutStartDate!,
                                end: workoutEndDate!,
                                workoutEvents: workoutEvents,
                                totalEnergyBurned: totalEnergyBurned,
                                totalDistance: totalDistance,
                                metadata:
[HKMetadataKeyIndoorWorkout:isIndoor]);//附加信息，元数据

        healthStore.save(workout) { success, _ in
            if success {
                //添加数据样本
                self.addSamples(toWorkout: workout)
            }
        }


    }

    private func addSamples(toWorkout workout: HKWorkout) {
      //总步数样本
        let totalCountSample =
HKQuantitySample(type:HKQuantityType.quantityType(forIdentifier:.
stepCount)!,
                                                quantity: totalCount,
                                                start: workoutStartDate!,
                                                end: workoutEndDate!)
        //平均心率样本
        let averageHeartRateSample = HKQuantitySample(type:
```

```
HKQuantityType.quantityType(forIdentifier:.heartRate)!,
                                            quantity: averageHeartRate,
                                            start: workoutStartDate!,
                                            end: workoutEndDate!)

        // 将两种数据样本添加到 workout 中
        healthStore.add([totalCountSample,averageHeartRateSample], to:
workout) { (success: Bool, error: Error?) in
            if success {
                // 添加成功
            }
        }
    }
```

反侵权盗版声明